

**XV**

# CL & CL

Computational Linguistics and Computer Languages

computer  
and  
automation  
institute  
hungarian  
academy  
of sciences

HELYBEN  
OLVASHATÓ

SZTE Egyetemi Könyvtár  
Egyetemi Gyűjtemény  
2



*COMPUTER AND AUTOMATION INSTITUTE  
HUNGARIAN ACADEMY OF SCIENCES*

**COMPUTATIONAL LINGUISTICS**

**AND**

**COMPUTER LANGUAGES**

**XV.**

ISSN 0324-2048

---

*Budapest, 1982.*

Editorial board:

**Bálint DÖMÖLKI (chairman)**

Theoretical Laboratory, Institute for Co-ordination of  
Computer Techniques

**Gábor DÁVID**

Computer and Automation Institute, Hungarian Academy  
of Sciences

**Ernő FARKAS (editor)**

Computer and Automation Institute, Hungarian Academy  
of Sciences

**Tamás GERGELY**

Research Institute for Applied Computer Sciences

**Tamás LEGENDI (editor)**

Research Group on Mathematical Logic and Theory of  
Automata, Hungarian Academy of Sciences

**Árpád MAKAI**

Research Group on Mathematical Logic and Theory of  
Automata, Hungarian Academy of Sciences

**Imre RUZSA**

University Eötvös, Budapest

**György SZÉPE**

Research Institute of Linguistics, Hungarian Academy  
of Sciences

**Dénes VARGA**

Hungarian Central Technical Library and Documentation  
Centre

Secretary to the board:

**Erzsébet CSUHAJ VARJÚ**

Computer and Automation Institute, Hungarian Academy of Sciences

Distributor for:

Albania, Bulgaria, China, Cuba, Czechoslovakia, German Democratic Republic,  
Korean People's Republic, Mongolia, Poland, Romania, U.S.S.R., Socialist Republic  
of Vietnam, Yugoslavia

**K U L T Ú R A**

Hungarian Trading Co. for Books and Newspapers

1389. Budapest,

P.O.B. 149, Hungary

For all other countries:

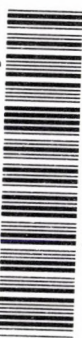
**JOHN BENJAMINS B.V.**

Periodical Trade

Amsteldijk 44

Amsterdam, Holland

SZTE Egyetemi Könyvtár



J000817369



**C 27196**

Responsible Publisher:

**Prof. Dr. TIBOR VÁMOS**

Director of the Computer and Automation  
Institute, Hungarian Academy of Sciences



CL      &      CL

## COMPUTATIONAL LINGUISTICS AND COMPUTER LANGUAGES

A scientific periodical published in English under the auspices of the  
COMPUTER AND AUTOMATION INSTITUTE, HUNGARIAN ACADEMY OF SCIENCES,

### *Topics of the periodical:*

The editorial board intends to include papers dealing with the syntactic and semantic characteristics of languages relating to mathematics and computer science, primarily those of summarizing, surveying, and evaluating, i.e. novel applications of new results and developed methods.

Papers under the heading of "Computational Linguistics" should contribute to the solution of theoretical problems on formal handling and structural relations of natural languages and to the researches on formalization of semantics problems, inspired by computer science.

Papers under the heading of "Computer Languages" should analyse problems of computer science primarily from the point of view of means of man-machine communication. For example it includes methods of mathematical logic, examining problems on formal contents and model theory of languages.

The periodical is published twice a year in December and June. Deadlines are 28 February and 31 August.

All correspondence should be addressed to:

COMPUTER AND AUTOMATION INSTITUTE  
HUNGARIAN ACADEMY OF SCIENCES  
*Scientific Secretariat*  
1502 Budapest  
P.O.B. 63.

### Subscription information:

Available from: JOHN BENJAMINS BV.  
Periodical Trade  
Amsteldijk 44 Amsterdam (Z)  
HOLLAND



## NOTES FOR AUTHORS

Original papers only will be considered. Manuscripts are accepted for review with the understanding that all persons listed as authors have given their approval for the submission of the paper; further, that any person cited as a source of personal communications has approved such citation.

Manuscripts should be typed in double spacing on one side of A4 (210 x 297 mm) paper, and authors are urged to aim at absolute clarity of meaning and an attractive presentation of their texts. Each paper should be preceded by a brief abstract in a form suitable for reproduction in abstracting journals.

The abstract should consist of short, direct, and complete sentences. Typically, its length might be 150 to 200 words. It should be informative enough to serve in some cases as a substitute for reading the paper itself. For this reason, the abstract should state the objectives of the works, summarize the results, and give the principal conclusions and recommendations. It should state clearly whether the focus is on theoretical developments or on practical questions, and whether subject matter or method is emphasized. The title need not be repeated. Work planned but not done should not be described in the abstract. Because abstracts are extracted from a paper and used separately, do not use the first person, do not display mathematics, and do not use citation reference numbers.

Number each page. Page 1 should contain the article title, author and coauthor names, and complete affiliation(s) (name of institution city, state, and zip code). At the bottom of page 1 place any footnotes to the title (indicated by superscript <sup>+</sup>, <sup>+</sup>, <sup>±</sup>). Page 2 should contain a proposed running head (abbreviated form of the title) of less than 35 characters. References should be listed at the end in alphabetical order of authors and should be cited in the text in forms of author's name and date.

Diagrams should be in Indian ink on white card or on cloth. Lettering should conform to the best draughtsmanship standards, otherwise it should be in soft pencil. Captions should be typed on a separate sheet. Particular care should be taken in preparing drawings; delay in publication results if these have to be redrawn in a form suitable for reproduction. Photographs for half-tone reproduction should be in the form of highly glazed prints.

List of Symbols. Attach to the manuscripts a complete typewritten list of symbols, identified typographically, not mathematically. This list will not appear in print but is essential in order to avoid costly author's corrections in proof (If equations are handwritten in the text then the list of symbols should also be handwritten.) Distinguish between "oh." "zero" "el." "on": "kappa." "kay": upper and lower case "kay"; etc. Indicate also when special type is required (German, Greek, vector, scalar, script, etc.); all other letters will be set in italic.

Authors are themselves responsible for obtaining the necessary permission to reproduce copyright material from other sources.



# C O N T E N T

	Page
F. BERMAN: Compactness in Models of Propositional Dynamic Logic . . .	7
L. BÖSZÖRMÉNYI: MODULA-2 Used in the Implementation of a Virtual Terminal Model . . . . .	21
S.D. COMER: Inductive Domains and Algebraic Semantics of CF Languages	43
E. CSUHAJ VARJÚ: Some Basic Properties of k-bounded Internations of Grammar Forms . . . . .	51
G. DÁVID: On the Basic Concepts of SDS (System Development System) Part II. . . . .	75
Z. ÉSIK: On Generalized Iterative Algebraic Theories . . . . .	95
I. FUTÓ and J. SZEREDI: A Very High Level Discrete Simulation System T-Prolog . . . . .	111
L. GEREVICH: A Parsing Method Based on Van Wijngaarden Grammars . . .	133
T. GERGELY and L. ÚRY: Representation and Verification of Communicating Sequential Processes . . . . .	157
T. GYIMÓTHY and J. DOMBI: Syntactic Pattern Recognition with Modified Fuzzy Automata . . . . .	175
G. PRÓSZÉKY, Z. KISS and L. TÓTH: Morphological and Morphonological Analysis of Hungarian Word-Forms by Computer . . . . .	195
E. SIMON: Languages Design Objectives and the Change System . . . . .	229
M. A. SUCHENEK: Effective Logic of Programming Languages . . . . .	249





## COMPACTNESS IN MODELS OF PROPOSITIONAL DYNAMIC LOGIC\*

Francine Berman  
Purdue University  
West Lafayette  
U S A

### Section 1: INTRODUCTION AND PREREQUISITES

Propositional Dynamic Logic (PDL) is a formal language for reasoning about programs. It was first developed by Fischer and Ladner [F&L] as a simple yet elegant programming logic for which the satisfiability problem was decidable. In this paper, we discuss compactness for several classes of models of PDL. We show that the broader interpretation of looping programs in nonstandard models of PDL admits compactness whereas the stricter interpretation of loops in Standard models does not. In Section 3, we use these results to explore compactness in Dynamic Algebra.

Elements of Propositional Dynamic Logic are programs and formulas. Programs are regular expressions with tests. Formulas are boolean combinations of simpler formulas and programs combined with modal operators. For a detailed explanation of the language, we refer the reader to [F&L] or [Be].

Definition: A model of PDL is a triple  $(W, \Pi, \rho)$  in which

$W$  is a set of states,  
 $\Pi$  is a formula valuation function, and  
 $\rho$  is a program valuation function.

$\Pi$  is constrained so that

$$\Pi(\neg p) = W - \Pi(p)$$

$$\Pi(p \vee q) = \Pi(p) \cup \Pi(q)$$

$$\Pi(\langle a \rangle p) = \{w \mid \exists v ((w, v) \text{ is in } \rho(a) \text{ and } v \text{ is in } \Pi(p))\}$$

$$\Pi([a]p) = \Pi(\neg \langle a \rangle \neg p)$$

\* This research was supported by NSF Grants MCS77-02474 and MCS80-05387.

$\rho$  is constrained so that

$$\rho(p?) = \{(w, w) \mid w \text{ is in } \Pi(p)\}.$$

We say that a formula  $p$  is satisfiable in model  $M$  iff there is a state  $w$  in  $M$  such that  $w$  is in  $\Pi_M(p)$ . Formula  $p$  is satisfiable in a class of models  $K$  iff there is some model  $M$  in  $K$  such that  $p$  is satisfiable in  $M$ . We say that  $p$  is valid in  $M$  (or a class of models  $K$ ) iff  $\neg p$  is not satisfiable in  $M$  (or  $K$ ).

Definition: An  $S$ -sound model is a model in which a set of schemas  $S$  is valid.

Definition: A Loop Invariant model is a model in which  $\rho$  is constrained so that

$$\rho(a; b) = \rho(a) \cdot \rho(b)$$

$$\rho(a \cup b) = \rho(a) \cup \rho(b)$$

$$\rho(a^*) \text{ contains } \rho(\text{true?}) \cup \rho(a) \text{ and is transitive.}$$

Definition: A Standard model is a Loop Invariant model in which  $\rho(a^*) = \cup \rho(a^n)$ .

Definition: Let  $D$  denote the proof system with rules of inference

Modus Ponens:  $A$  and  $A \rightarrow B$  implies  $B$ ,

Necessitation:  $A$  implies  $[b]A$ ,

and axiom schemas

all propositional tautologies

$$[a](p \rightarrow q) \rightarrow ([a]p \rightarrow [a]q)$$

$$[a](p \wedge q) = [a]p \wedge [a]q$$

$$\langle a; b \rangle p = \langle a \rangle \langle b \rangle p$$

$$\langle a \cup b \rangle p = \langle a \rangle p \vee \langle b \rangle p$$

$$\langle p? \rangle q = p \wedge q$$

$$p \rightarrow \langle a^* \rangle p$$

$$\langle a \rangle p \rightarrow \langle a^* \rangle p$$



$$\begin{aligned} <a^*> <a^*>p \dashv\vdash <a^*>p \\ <a^*>p \dashv\vdash (\neg p \vee <a^*>(\neg p \wedge <a>p)) \end{aligned}$$

Following the classical definitions, a formula  $p$  is provable ( $\vdash p$ ) if there is a finite proof in which each line either is an axiom or follows from the preceding lines by the rules of inference. A finite set of formulas  $\{p_i\}$  is consistent if the negation of their conjunction is not provable ( $\not\vdash \neg(\wedge p_i)$ ). A set of formulas is consistent if every finite subset is consistent. Sets which are not consistent are inconsistent.

Let  $\text{Pr}(S)$  be the set of provable formulas of a deductive system with axiom schemas  $S$  and rules of inference Necessitation and Modus Ponens. For a class of models  $M$ , let  $\text{Th}(M)$  be the set of formulas valid in every model of  $M$ . A class of models  $M$  is complete with respect to a proof system with axiom schemas  $S$  if  $\text{Th}(M) = \text{Pr}(S)$ . A set of models  $M$  is compact if for every set of formulas  $T$ ,  $T$  is simultaneously satisfiable in  $M$  iff every finite subset of  $T$  is simultaneously satisfiable in  $M$ .

The classes of models defined in this section are closely related. For  $S=D$ , the classes of D-sound models, Loop Invariant models and Standard models are indistinguishable by PDL expressions [Be]. Consequently, completeness for any one of these classes of models with respect to  $D$  implies completeness for the other classes. Although, all three classes of models can be shown to be complete directly, we show in Section 2 that the proof of completeness for D-sound models provides a technique which can be used to show that the classes of Loop Invariant and D-sound models are compact.

## Section 2: COMPACTNESS

First, we show the class of D-sound models to be complete with respect to the proof system  $D$ . As a consequence of techniques in the completeness proof, we will be able to show the classes of D-sound and Loop Invariant models to be compact





with respect to D. The proof of completeness is a Henkin argument in which a canonical model  $M_0$  is defined whose states are maximal consistent sets of formulas. Consistency is equivalent to satisfiability within this model. The proof is similar to that given by Segerberg for classical modal systems [Seg].

### Theorem 1

The class of D-sound models is complete with respect to the proof system with axiom schemas D.

### Proof of Theorem 1

Let  $M_0$  be the model in which

- $W = \{w \mid$  1,  $w$  is a consistent set of formulas,  
2) for each formula  $p$ ,  $p$  is in  $w$  or  $\neg p$  is in  $w$ ,  
3, if  $p$  and  $p \rightarrow q$  are in  $w$  then  $q$  is in  $w$ ,  
4)  $\text{Pr}(D)$  is a subset of  $w\}$

$\Pi(p) = \{v \mid p \text{ is in } w\}$  for all basic formulas  $p$

$\rho(a) = \{(w, v) \mid \text{for all formulas } p, [a]p \text{ is in } w \text{ implies } p \text{ is in } v\}$  for any program  $a$ .

Extend  $\Pi$  and  $\rho$  in the usual way so that  $M_0$  is a model. We show that a formula is satisfiable in the class of D-sound models iff it is satisfiable in  $M_0$ . Then we show that the set of valid formulas of  $M_0$  is  $\text{Pr}(D)$ . Completeness follows.

### Lemma 1

For all formulas  $p$  and for all states  $w$  in  $M_0$ ,

$w$  is in  $\Pi_{M_0}(p)$  iff  $p$  is in  $w$ .

### Proof of Lemma 1

For notational convenience, we will drop the subscript on  $\Pi$ . Proceed by induction on the length of  $p$  as a sequence of symbols.

If  $p$  is a basic formula then by definition,  $w$  is in  $\Pi(p)$  iff  $p$  is in  $w$ .

For the induction step, assume the result is true for all formulas  $p$  with  $|p| < n$ . Consider  $|p| = n$ .

If  $p$  is of the form  $qvr$  or  $\neg q$  then the result follows by induction and conditions 2) and 3) of the definition of  $W$ .

The nontrivial case is when  $p = [a]q$ . For one direction, assume towards a contradiction that  $w$  is in  $\Pi([a]q)$  and  $\neg[a]q$  is in  $w$ .

Consider the set  $S = \{r \mid [a]r \text{ is in } w\}$ . We wish to show that  $S$  is consistent. Assume not. Then there exist formulas  $r_1, \dots, r_n$  such that

$\vdash r_1 \wedge \dots \wedge r_n \rightarrow \text{false}$  but then

$\vdash [a](r_1 \wedge \dots \wedge r_n \rightarrow \text{false})$  and

$\vdash [a] r_1 \wedge \dots \wedge [a] r_n \rightarrow [a] \text{false}.$

$\vdash [a] \text{false} \rightarrow [a]q$  so

$\vdash [a] r_1 \wedge \dots \wedge [a] r_n \rightarrow [a]q$  and  $[a] r_1 \wedge \dots \wedge [a] r_n \rightarrow [a]q$  is in  $w$ .

This provides a contradiction since the  $[a] r_1 \wedge \dots \wedge [a] r_n$  and  $\neg[a]q$  are in  $w$ .

Now consider a slightly larger set  $T = S \cup \{\neg q\}$ . We have just shown  $S$  to be consistent. We would like to show that the addition of  $\neg q$  preserves consistency. The proof of this is similar to the preceding argument and left to the reader.

That  $T$  is consistent with respect to  $D$  implies that  $\text{TUPr}(D)$  is consistent. As a consequence of Lindenbaum's theorem we can extend  $\text{TUPr}(D)$  to a maximal consistent set of formulas  $v$ . By



construction,  $v$  is in  $W$  and also  $(w, v)$  is in  $\rho(a)$ . Recall that by hypothesis,  $w$  is in  $\Pi([a]q)$ , hence  $v$  is in  $\Pi(q)$ . But by construction,  $\neg q$  is in  $v$  and by induction,  $v$  is in  $\Pi(\neg q)$ . This is a contradiction.

We have shown that  $w$  in  $\Pi([a]q)$  implies that  $[a]q$  is in  $w$ . The other direction is a straightforward proof by contradiction using the definitions.  $\square$

### Corollary

Let  $p$  be a formula. Then  $p$  is satisfiable in the class of D-sound models iff  $p$  is satisfiable in  $M_0$ .

### Proof of Corollary

Let  $p$  be a formula. Assume that  $p$  is satisfiable at a state  $w$  in a D-sound model  $M$ . Let  $S_w$  be the set of formulas satisfiable at  $w$  in  $M$ . By definition,  $S_w$  is a maximal consistent set of formulas. Hence  $S_w$  is a state in  $M_0$ . By Lemma 1,  $p$  in  $S_w$  implies that  $S_w$  is in  $\Pi(p)$  in  $M_0$ . Hence,  $p$  is satisfiable in  $M_0$ .

Conversely, assume that  $p$  is satisfiable in  $M_0$ . By Lemma 1 and construction,  $M_0$  is a D-sound model since  $D$  is contained in  $\text{Pr}(D)$  and  $\text{Pr}(D)$  is contained in every state. Hence  $p$  is satisfiable in some D-sound model.  $\square$

### Lemma 2

$$\text{Th}(M_0) = \text{Pr}(D)$$

### Proof of Lemma 2

For one direction, it suffices to notice that since  $\text{Pr}(D)$  is contained in each state in  $M_0$ ,  $\text{Pr}(D)$  must be contained in  $\text{Th}(M_0)$ .

For the other direction, let  $p$  be a formula in  $\text{Th}(M_0)$  but not in  $\text{Pr}(D)$ . Then  $p$  is not provable in  $D$ . By definition,  $\neg p$



is consistent with  $D$ . In particular, the set  $\text{Pr}(D) \cup \{-p\}$  can be extended to a maximal consistent set  $w$ . By construction,  $w$  is a state in  $M_0$ . This provides a contradiction since then both  $p$  and  $\neg p$  are satisfiable at  $w$ .  $\square$

#### Proof of Theorem 1 continued

By Lemma 1,  $\text{Th}(M_0) = \text{Th}(D\text{-sound models})$ . By Lemma 2,  $\text{Th}(M_0) = \text{Pr}(D)$ . Hence the class of  $D$ -sound models is complete with respect to  $D$ .  $\square$

With this construction, it is now simple to show that the class of  $D$ -sound models is compact.

#### Theorem 2

The class of  $D$ -sound models is compact.

#### Proof of Theorem 2

Let  $S$  be a set of formulas. Clearly, if  $S$  is satisfiable in a  $D$ -sound model then every finite subset of  $S$  is satisfiable in a  $D$ -sound model.

Conversely, suppose that every finite subset of  $S$  is satisfiable in some  $D$ -sound model. In particular, this implies that every finite subset of  $S$  is consistent. Hence  $S$  is a consistent set of formulas. By Lindenbaum's theorem we can extend  $S$  to a maximal consistent set of formulas  $w$  containing  $\text{Pr}(D)$ . By construction,  $w$  is a state in  $M_0$ . By Lemma 1,  $S$  is satisfiable at  $w$  in  $M_0$ . Hence  $S$  is simultaneously satisfiable in the class of  $D$ -sound models.  $\square$

To show that the class of Loop Invariant models are also compact, we can use this technique in conjunction with a theorem from [Be]. The theorem states



- 1) Every Loop Invariant model is a D-sound model.
- and
- 2) Every D-sound model  $M$  can be extended to a Loop Invariant model  $M'$  such that for all formulas  $p$  and all states  $w$  in  $M$ ,  
 $w$  is in  $\Pi_M(p)$  iff  $w$  is in  $\Pi_{M'}(p)$ .

### Theorem 3

The class of Loop Invariant models is compact.

### Proof of Theorem 3

Let  $S$  be a set of formulas. As before, if  $S$  is simultaneously satisfiable in the class of Loop Invariant models then every finite subset of  $S$  is satisfiable in the class of Loop Invariant models.

Conversely, assume that every finite subset of  $S$  is satisfiable in the class of Loop Invariant models. As in the proof of Theorem 2, this implies that  $S$  is consistent. In addition,  $S$  is simultaneously satisfiable at some state  $w$  in the D-sound model  $M_0$ . By part 2) of the theorem described above,  $M_0$  can be extended to a Loop Invariant model  $M_0'$  in which every formula satisfiable at  $w$  in  $M_0$  is satisfiable at  $w$  in  $M_0'$ . In particular, this is true for each of the formulas in  $S$ . Hence  $S$  is simultaneously satisfiable in the Loop Invariant model  $M_0'$ .  $\square$

Unfortunately this technique does not work for the class of Standard models. The set of formulas  $A = \{[a^n]p \mid n \geq 0\} \cup \{\langle a^* \rangle - p\}$  is finitely satisfiable but not simultaneously satisfiable in the class of Standard models. The reason for this discrepancy is that although loops are expressed equivalently in the classes of Standard, Loop Invariant and D-sound models, their termination is interpreted differently in each of these classes. In Standard models, all loops are assumed to terminate success-



fully, i.e. upon fulfillment of their exit conditions and after a finite number of iterations of the loop. In Loop Invariant and D-sound models, loops may terminate unsuccessfully as well, i.e. through the exceeding of time or space limitations.

### Section 3: COMPACTNESS IN DYNAMIC ALGEBRA

Dynamic Algebras were developed by Kozen [K] and Pratt [P]. The interpretation of PDL within the algebraic setting has the advantage of a clean exposition and simple, yet elegant proofs. For simplicity, we will consider only test-free (without ?) models of PDL in this section. The addition of ? complicates the proofs somewhat but does not change the results. For completeness, we will give the basic definitions from Dynamic Algebra. For a fuller treatment see [P] or [K], additional material can be found in [N].

Definition (Kozen): A ( $*$ -continuous) dynamic algebra is a two-sorted algebra  $(K, B, \langle \rangle)$  where  $K$  is a regular algebra (with  $a^* = \sup a^n$ ),  $B$  is a boolean algebra, and  $\langle \rangle: K \times B \rightarrow B$  is a scalar multiplication satisfying the equations

$$\begin{aligned} \langle a \rangle 0 &= 0 \\ \langle a \rangle (p \vee q) &= \langle a \rangle p \vee \langle a \rangle q \\ \langle a \cup b \rangle p &= \langle a \rangle p \vee \langle b \rangle p \\ \langle ab \rangle p &= \langle a \rangle \langle b \rangle p \\ \langle 1 \rangle X &= X \\ \sup \langle a^n \rangle X &= \langle a^* \rangle X. \end{aligned}$$

Definition: A dynamic algebra is separable if for each  $a \neq b$  in  $K$ , there exists some  $X$  in  $B$  with  $\langle a \rangle X \neq \langle b \rangle X$ .

Definition: Let  $(S, K, B)$  be a structure where

$S$  is a set of states,

$K$  is a regular algebra of binary relations on  $S$  (under the usual set theoretic operations),



$\langle a \rangle X = \{s \mid \exists v((s, v) \text{ is in } a \text{ and } v \text{ is in } X)\}$  is in B  
for every  $a$  in  $K$  and  $X$  in  $B$ .

Then  $(K, B, \langle \rangle)$  is a dynamic algebra called a Kripke model.

Definition: A standard Kripke model is a Kripke model in which  $a^* = \bigcup a^n$  for every  $a$  in  $K$  (i.e. for every  $a$  in  $K$ ,  $\sup a^n = \bigcup a^n$  is in  $K$ ).

Every Loop Invariant model  $M = (W, \Pi, \rho)$  can be identified with a dynamic algebra  $(K, B, \langle \rangle)$  in which  $K$  is a regular algebra with domain  $K = \{\rho(a) \mid a \text{ is a program}\}$  and  $B$  is a boolean algebra with domain  $B = \{\Pi(p) \mid p \text{ is a PDL formula}\}$ . Note that  $\rho(a^*) = \sup \rho(a^n)$  but that this may only properly contain  $\bigcup \rho(a^n)$ .

Conversely, every Kripke model can be identified with a Loop Invariant model by letting each element of  $K$  represent the interpretation of some program and letting each element of  $B$  represent the interpretation of some formula.

There is a one-to-one correspondence between the class of standard Kripke models and the class of Standard models defined in Section 2. Note that there are separable \*-continuous dynamic algebras which are not represented by any standard Kripke model [K]. This parallels the fact that there are Loop Invariant models which are not Standard [Be].

In [K], Kozen showed that every separable dynamic algebra is isomorphic to a Kripke model. We can use this result in defining compactness for sets of separable dynamic algebras.

Definition: A set of separable dynamic algebras is compact iff the set of Loop Invariant models associated with its set of isomorphic Kripke models is compact.



Theorem 4

The class of all Kripke models is compact.

Proof of Theorem 4

By the preceding remarks, there is a one-to-one correspondence between the class of Loop Invariant models and the class of Kripke models. Hence the compactness of the class of Loop Invariant models implies the compactness of the class of Kripke models.  $\square$

Corollary

The class of separable dynamic algebras is compact.

Proof of Corollary

By Kozen's results, every separable dynamic algebra is isomorphic to a Kripke model. Compactness follows by Theorem 4.  $\square$

Theorem 5

Let  $C$  be a class of standard Kripke models. If there exists a program  $a$  and a formula  $p$  such that the set of formulas  $\{[a^n]p \mid n \geq 0\} \cup \{ \langle a^* \rangle - p \}$  is finitely satisfiable then  $C$  is not compact.

Proof of Theorem 5

It is straightforward to see that the set of formulas  $\{[a^n]p \mid n \geq 0\} \cup \{ \langle a^* \rangle - p \}$  is not simultaneously satisfiable in any Standard model for any program  $a$  and formula  $p$ . Hence this set is not simultaneously satisfiable in any standard Kripke model for any  $a$  and  $p$ . If  $C$  is compact then any set of



formulas is simultaneously satisfiable iff it is finitely satisfiable. Consequently if a set of formulas  $\{[a^n]p \mid n \geq 0\} \cup \{ \langle a^* \rangle - p \}$  is finitely satisfiable in  $C$  then  $C$  cannot be compact.  $\square$

Note that a consequence of Theorem 5 is that the class of standard Kripke models is not compact.

### Section 3: SUMMARY

In this paper, we have explored compactness for several classes of models of Propositional Dynamic Logic. We showed that the nonstandard classes of D-sound and Loop Invariant models are compact using a classical Henkin technique. In contrast, the class of Standard models is not compact.

In Section 3, we extended compactness to Dynamic Algebra and characterized compactness for the class of Kripke models. We also gave a sufficient condition for the noncompactness of standard Kripke models.

### ACKNOWLEDGEMENTS

We would like to thank Mike Fischer and Miriam Lucian for several helpful discussions on this material. We would also like to thank Jan Cuny for her careful reading of this paper.

### BIBLIOGRAPHY

- [Be] Berman, F., "Semantics of Looping Programs in Propositional Dynamic Logic," submitted to Mathematical Systems Theory.
- [F&L] Fischer, M. and R. Ladner, "Propositional Dynamic Logic of Programs," JCSS 18:2, April 1979.



- [K] Kozen, D., "A Representation Theorem for Models of \*-Free PDL," Proc. 7th Intl. Colloq. on Automata, Languages and Programming, Lecture Notes in Computer Science 85, Springer-Verlag, Berlin 1980.
- [N] Némethi, I., "Dynamic Algebras of Programs," Fundamentals of Computation Theory 1981, Lecture Notes in Computer Science 117, Springer-Verlag, Berlin 1981, pp.281-290.
- [P] Pratt, V., "Dynamic Algebras and the Nature of Induction," Proc. 12th ACM Symposium on the Theory of Computing, May 1980.
- [Seg] Segerberg, K., "An Essay in Classical Modal Logic," (Volume 1), Ph.D. Dissertation, Uppsala Universitet, 1971.





## MODULA-2 USED IN THE IMPLEMENTATION OF A VIRTUAL TERMINAL MODEL

*László Böszörményi*

Computer and Automation Institute  
Hungarian Academy of Sciences  
Budapest, Hungary

### SUMMARY

This paper is meant to report on the experiences gained by using the language Modula-2 for implementing a Virtual Terminal Model. It touches some questions of the virtual terminal technique, but its main concern is the use of Modula-2 in a real application. We show, how Modula-2 has supported and influenced our choices in questions like decomposition, handling of parallelities etc. The paper contains parts of the listings of the model. The entire listing is available at the author.

KEY WORDS: Programming-language Module Decomposition  
Computer-network Virtual-Terminal Formal-specification.

### INTRODUCTION

The main goal of this paper is to report on the experiences, gained with Modula-2 at the implementation of a Virtual Terminal Model. The Virtual Terminal Model serves as a tool for experimenting with the virtual terminal functions of a computer network, and also a formal specification for different groups of implementors.

A computer network may be regarded as a complex of communication media and of several users being connected to it. The users may be both programs or humans. Networks are generally built up as hierarchical systems of layers. Each layer may be regarded as a communication media again, to which users may be connected.



A layer communicates with the neighbouring layers via interfaces /also called local protocols/. Each layer uses the services of the underlying layer, and offers a new level of services to its users.

Components of a layer are called to be remote, if they communicate with each other, using the services of the underlying layer. The set of rules of this kind of communication is called protocol /or remote protocol/.

In the definition of a layer it is necessary to give both the remote and the local protocol definitions. In many cases solely one of these is given, sometimes they are confused. It is important to distinguish between the external /interface/ and the internal /protocol/ operation of a layer.

The implementation of a layer usually happens on different systems by different groups of people. This fact raises an extremely strict demand on the definition. This is, what influenced us to develop a model in a high-level language, which has the expressive power necessary for a formal specification. In an ideal case, we would have had the appropriate compilers for all participant systems of the network. It was, however, not our case, and it seemed to be a too great effort to implement Modula-2 compilers for all participants.

A similar project has been made by A. Dünki and P. Schicker /9/. They gave a formal implementation of a virtual terminal in Pascal. We have learned a lot from the works of G. Bochmann and T. Joachim as well /10,11/. They developed a real implementation of the X.25 layers in Concurrent Pascal. They also worked out several principles of decomposition of a protocol, which method has influenced us in a considerable degree.

We have used the Modula-2 compiler written in Zürich at the Institut für Informatik, ETH. The Compiler was running on a PDP 11/34.



### Modula-2's aid for decomposition.

Modula-2 offers the module concept to assist the programmer in decomposing his problem. The Modula language has already had this feature /2/, /4/. The module is a syntactical unit, that hides the module's inner objects, except the explicitly exported ones, and lets the module's environment unknown except the explicitly imported objects. The module is a convenient tool for expressing abstract data structures, it has a "forcing power" for a higher degree of modularity and hence for a higher reliability as well.

Modula-2 extends the module concept with a very important new feature. This is the ability to divide a module into two parts, into the definition part, that describes the exported objects only, and the implementation part /1/. The definition module may contain constant, type and variable declarations, and the head of the procedures to be exported. The body of these procedures are implemented in the implementation module. Several different implementations may be written for the same definition. With the definition/implementation module concept Modula-2 has also created a conceptual perfect way for the separate /but not independent/ compilation of modules /5/.

Modula-2 /like Modula or Mesa/ regards the module concept preferably as a general information hiding tool, in contrast with languages like Simula or Concurrent Pascal, that regard it as a data abstraction mechanism /4/. As a consequence, Modula-2 does not support the multiplication of modules. Wirth states, that usually only one instance of modules exists/3/. Unfortunately, this statement does not stand for us, but rather the opposite of it. In the networking area there is nothing more typical, than having several instances of relatively large modules. The possibility to start processes in several incarnations does not cover this demand. Several instances of virtual circuits or virtual terminals etc. could have been expressed as instances of modules most conveniently. In Modula-2, the existence of several instances must be reflected in the data structure of the



module. Actually: the variables local to a module /but not to a procedure/, that should have several instances, must be declared as arrays of their original type. If we use Modula-2 also as a specification language, it seems to be rather disturbing to have, at all variables local to a module, one dimension more. To avoid it, we have written all modules supposing only one instance in the first step, and we have replaced the mentioned variables in a second step. It was a rather long and dull editing-job.

### Parallel activities in Modula-2

Modula-2 does not contain an integrated process-scheduler, like many similar languages do /Modula /2/, Concurrent Pascal /6/ etc./. Instead, it offers a few, low-level primitives, whereby it encourages the programmer to design his scheduler of his own. The concept of Modula-2 is, that the high-level language should not support a given scheduling strategy, but should rather support the programmer in developing any kind of scheduler. If the programmer does not want to design a scheduler of his own, he may use any other schedulers written by others. Examples for schedulers are given by J.Hoppe /7/, where a message-oriented-solution is presented, and by Wirth /1/, where a Modula-2 implementation of the original Modula scheduler /2/ is shown.

Modula-2 does not support processes as sequential activities competing for some common resources in a true parallel way, it supports rather coroutines /nevertheless, also called processes/ running on a single processor that never loose the processor once obtained, except explicitly required. Normal processes run on the lowest hardware priority level. Device processes run on the level of the appropriate device. They must never call a procedure declared in a module of lower level. This rule guarantees, that processes work as coroutines.



For our virtual terminal model we have applied the Modula-2 implementation of the original Modula scheduler /1/ with some slight changes. The scheduler defines and exports the type signal /implemented as a pointer/ and a related set of operations:

- WAIT/S/: Causes the running process to enter the waiting condition and lets run the next ready process;
- SENDDOWN/S/: Causes the first process waiting for S enter the ready condition. It is ineffective, if there is no such process;
- SEND/S/: Similar to SENDDOWN, but it also starts immediately the process which has just been made ready;
- AWAITED/S/: A Boolean function, returns true, if one or more processes are waiting for S.

The scheduler also exports the procedure PAUSE for delaying a process, the procedure DOIO for awaiting the completion of an I/O operation, and the procedure STARTPROCESS to start any global, parameterless procedure as a coroutine, possessing an own data space and concurring with the other coroutines.

Interrupts are handled in a nonpreemptive way, i.e. after an interrupt has occurred, the interrupted process is marked, to be resumed as soon as possible. It implies, that the user should avoid the use of SEND in a device process, as it may cause to start a chain of processes until the interrupted process may be resumed. Instead, we use SENDDOWN in device processes.

### The Virtual Terminal Layer

The task of the Virtual Terminal Layer of a computer network is to offer a homogenous terminal image throughout the network, independent of the physical terminal devices.



The underlying layer is often called Transport Service or Transport Layer. We assume, that it is able to transport blocks of information /often called letters/ in a proper sequence, free of error, with a finite speed. We also assume, that it is able to transport short /one byte/ blocks /often called telegrams or interrupts/, independent of the flow of other blocks. The Transport Layer will also offer tools for opening and closing connections /often called liaisons/ between two remote users of it. Our model, however, does not contain elements dealing with connecting and disconnecting. We consider these tasks belonging to a separate unit. Thus, the Virtual Terminal Model expresses a connected state.

The users of the Virtual Terminal Layer may be both programs and humans. Our model defines a user interface, that satisfies this requirement. The implementation of the user interface, however, is in accordance with a human user operating a physical device. The implementation of the user interface to application programs will be dependent on the application itself. Such interface modules may easily be integrated into our model.

The place of the Virtual Terminal Layer is shown on figure 1.

#### The Virtual Terminal Model.

The most important step of the design has been the decomposition of the model. The flexibility of the model is of primary importance. In the decomposition we have relied on the principles worked out by Bochmann and Joachim /11/. We have also transformed our components into an implementation in terms of processes, monitors and classes. Nevertheless, there is no syntactical difference between monitors and classes, they are both expressed in the form of modules. The components were chosen so, that they should build functionally separate /although not independent/ units. State variables are always owned by different modules and



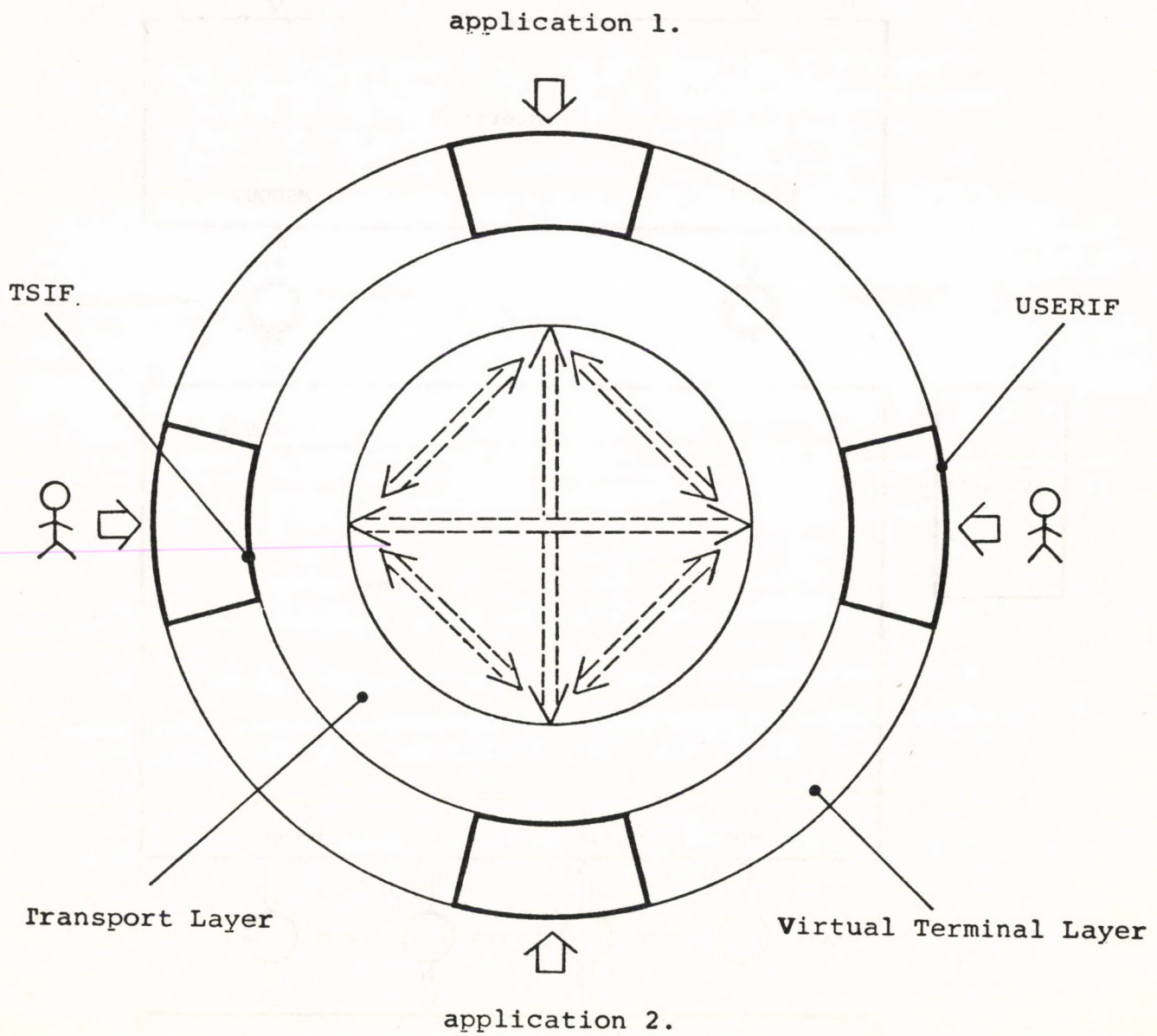


FIGURE 1.

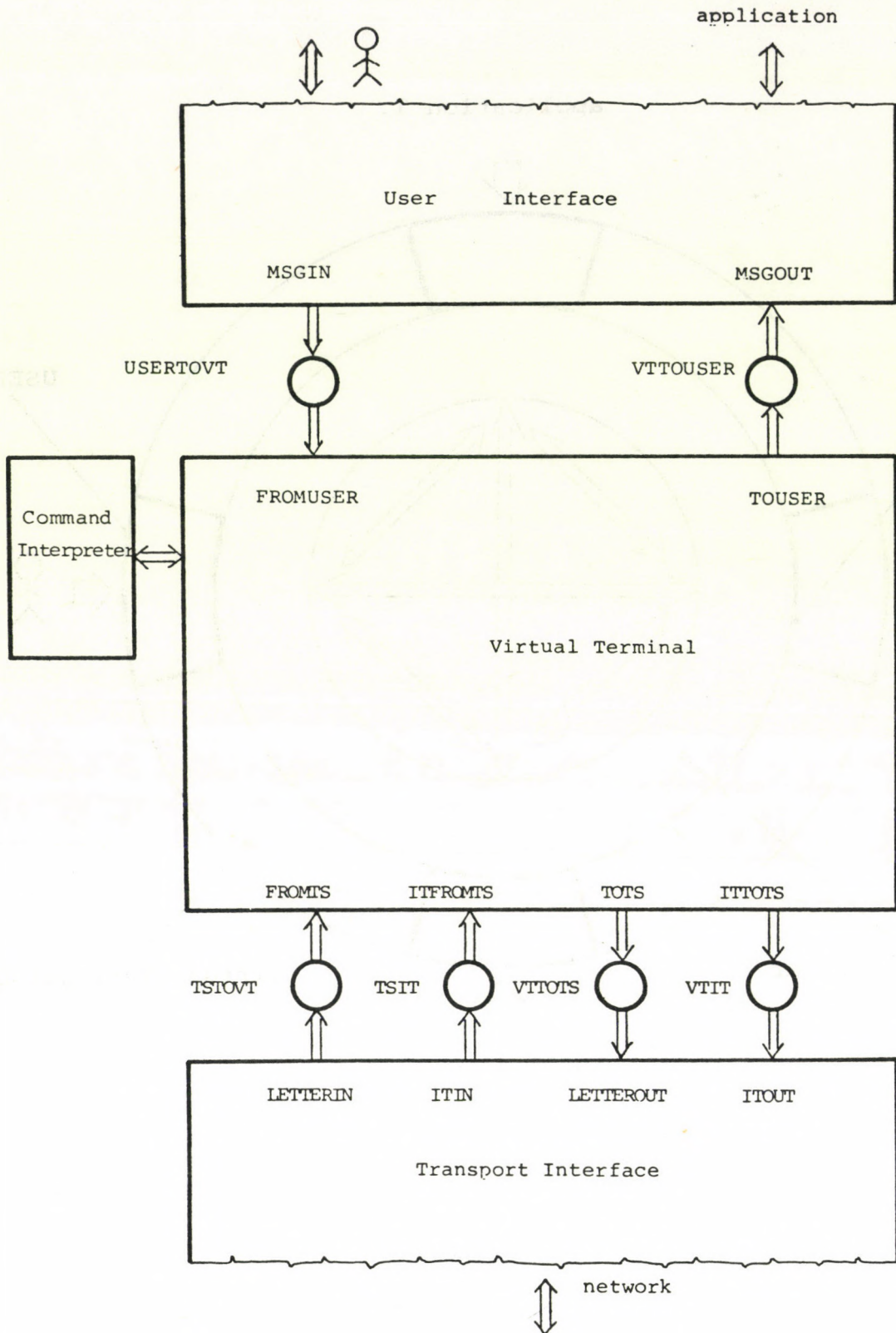


FIGURE 2.



hidden from external influences. "Active" events are represented by processes. All processes are gathered into one single module and the other modules of the model are all "passive". The processes work on a producer/consumer basis.

Our Virtual Terminal Model has three main components:

- The User Interface /USERIF/ module;
- The Transport Interface /TSIF/ module;
- The Virtual Terminal Protocol /VT/ module.

The structure of the Virtual Terminal Model is shown on figure 2.

The main components may be well characterised by their definition module, i.e. by their interface with the environment.

The objects offered by the user interface to the virtual terminal is defined in the USERIF module:

```
DEFINITION MODULE USERIF;  
  
FROM BUFFER IMPORT BUFHEAD;  
EXPORT QUALIFIED  
  MSGIN,MSGOUT,MSGKIND,NL,STRTL,BREAK,KILLINE;  
  
CONST NL = 16C; (*CNTRL R*) STRTL = 23C; (*CNTRL S*)  
      BREAK = 33C; (*ESC*) KILLINE = 13C; (*CNTRL K*)  
  
TYPE MSGKIND = (PARTIAL,WITHEOM,WITHEOMYRT,KILLEDLINE);  
  
PROCEDURE MSGIN(VAR M: BUFHEAD; VAR MSGK: MSGKIND);  
  (*READS A MESSAGE FROM THE USER*)  
  
PROCEDURE MSGOUT(VAR M: BUFHEAD; MSGK: MSGKIND);  
  (*GIVES A MESSAGE TO THE USER*)  
  
END USERIF.
```

The user interface and the virtual terminal modules exchange information in messages. The message is a character stream that may contain any character. Some of them /those that are exported here, like NL, STRTL/ have a special meaning. Messages may be terminated in four ways:

- partial: continuation follows;
- with eom: normal termination;
- with eom and yourturn: gives the turn to the partner if alternating;
- killed line: the line is erroneous. This structure is reflected in the type MSGKIND. The module still exports two procedures: MSGIN for taking a message from the user and MSGOUT for giving a message to him.

The objects offered by the transport service to the virtual terminal are defined in the TSIF module:



```
DEFINITION MODULE TSIF;  
  
FROM BUFFER IMPORT BUFHEAD;  
EXPORT QUALIFIED  
  LETTERIN, LETTEROUT, ITIN, ITOUT, MAXBLOCKI;  
  
CONST MAXBLOCKI = 255;  
  
PROCEDURE LETTERIN(VAR L: BUFHEAD);  
  (*READS A LETTER FROM THE TS*)  
  
PROCEDURE LETTEROUT(VAR L: BUFHEAD);  
  (*GIVES A LETTER TO THE TS*)  
  
PROCEDURE ITIN(VAR IT: BUFHEAD);  
  (*READS AN INTERRUPT FROM THE TS*)  
  
PROCEDURE ITOUT(VAR IT: BUFHEAD);  
  (*GIVES AN INTERRUPT TO THE TS*)  
  
END TSIF.
```

The transport service and the virtual terminal modules exchange information in blocks /or letters/. Their formats are described in the Virtual Terminal Definition /12/. The TSIF module exports the maximum block length it is able to accept. It exports also procedures for information exchange. The model has two different implementations of these procedures. The first one replaces the transporting functions with procedures for displaying the outgoing and reading the incoming blocks and interrupts. It is a tool for experimenting with the virtual terminal, it enables us to look into the internal operation of the virtual terminal machine. We may follow the internal structure of information on the transport interface, and we also may simulate the incoming information. We may easily test the system's reaction on any kind of erroneous input. The second implementation module of the transport service is actually empty. It serves to connect two partners directly, through a dummy network. In both cases we have used two physical display units. In the first case one of them served the virtual terminal operator and the



other the test operator. In the second case both displays served virtual terminal operators. The structure allows us to exchange the implementation module for a true transport service module, thus turning our model into a true implementation.

The realisation of the Virtual Terminal Protocol is done in the VT module:

```
DEFINITION MODULE VT;

FROM BUFFER IMPORT BUFHEAD;
FROM USERIF IMPORT MSGKIND;
EXPORT QUALIFIED PARTYPES, ACTPAR, PARTOSET,
CODEPURGE, CODERESUME, CODEPLEASE, XSIZEDEFAULT,
CODEFREE, CODEMYTURN, CODEYOURTURN,
NOTICEKIND, SDNOT, SENDNOTICE, SENDIT, SENDSET, SHOWPARS,
FROMUSER, TOUSER,
ITTOTS, TOTS, ITFRONTTS, FRONTTS;

CONST CODEPLEASE = 360B; CODERESUME = 361B;
CODEPURGE = 351B;
CODEMYTURN = 2; CODEYOURTURN = 3;
CODEFREE = 0; XSIZEDEFAULT = 0;

TYPE
PARTYPES = (CLASS, AUXDEV, MODE, OVERPRINT, XSIZE);
PARAMETERS = ARRAY PARTYPES OF INTEGER;
NOTICEKIND = (NORMALTEXT, TEXTTURN,
CMDNOTFOUND, CMDUNDERSPEC, GOODCMD,
ITPROCESSING, SETPROCESSING,
ITREC, AGREEREC, DISAGREEREC,
AGREESENT, DISAGREESENT,
ACTPARAMS, MYLIMITS, PARTNERSLIMITS);

VAR ACTPAR, PARTOSET: PARAMETERS;

PROCEDURE SDNOT(NOTICE: NOTICEKIND);
(*SENDS A NOTICE TO THE USER*)

PROCEDURE SENDNOTICE(NOTICE: NOTICEKIND; PARAM: CHAR);
(*SENDS A NOTICE AND A PARAMETER TO THE USER*)

PROCEDURE SHOWPARS(VAR P: PARAMETERS; VAR B: BUFHEAD);
(*PUTS P-S COMPONENTS INTO B IN A READABLE FORM*)

PROCEDURE SENDIT(ITCODE: INTEGER);
(*SENDS AN INTERRUPT TO THE TS*)

PROCEDURE SENDSET;
(*SENDS A SET-ITEM IN A SINGLE BLOCK*)

PROCEDURE FROMUSER(VAR M: BUFHEAD; MSGK: MSGKIND);
(*READS A MESSAGE FROM THE USER*)
```



```
PROCEDURE TOUSER(VAR M: BUFHEAD; MSGK: MSGKIND);  
(*GIVES A MESSAGE TO THE USER*)
```

```
PROCEDURE ITTOTS(VAR IT: BUFHEAD);  
(*GIVES AN INTERRUPT FOR THE TS*)
```

```
PROCEDURE TOTS(VAR LETTER: BUFHEAD);  
(*GIVES A LETTER TO THE TS*)
```

```
PROCEDURE ITFRONTTS(VAR IT: BUFHEAD);  
(*READS A INTERRUPT FROM THE TS*)
```

```
PROCEDURE FRONTTS(VAR LETTER: BUFHEAD);  
(*READS A LETTER FROM THE TS*)
```

```
END VT.
```

The VT module exports constants, types, variables and procedures. Principally, it would be enough to export solely procedures, partly for the information exchange between the interfaces and partly for offering some services to the command interpreter. Nevertheless, the VT module also makes visible some parts of its data structure. These are used mainly by the command interpreter, and beside that, by the USERIF and TSIF in a strict read-only way.

The VT implementation module is divided into modules as well, according to the separate functions of the virtual terminal protocol /negotiation handling, interrupt handling, normal text handling/.

The interconnections and the data flows are represented by processes. There are two independent data flows between the VT and the USERIF modules /msgin and msgout/. There are four independent data flows between the VT and the TSIF modules /letterin, letterout, itin, itout/. The processes are kept as simple as possible. They connect two modules. They read a unit of information through an output procedure of the one module and they give it to the appropriate input procedure of the other module. We show the VTPRCS module, that contains all processes of the Virtual Terminal Model:



```
MODULE VTPRCB;  
(*VIRTUAL TERMINAL PROCESSES*)  
  
FROM SYSTEM IMPORT WORD;  
FROM PROCESSSCHEDULER IMPORT STARTPROCESS;  
IMPORT VT;  
IMPORT USERIF;  
IMPORT TSIF;  
  
TYPE WSP = ARRAY [0..250] OF WORD;  
  
VAR WSP1,WSP2,WSP3,WSP4,WSP5: WSP;  
  
PROCEDURE USERTOVT;  
(*DATA FLOW FROM THE USER TO THE VIRTUAL TERMINAL*)  
VAR M: BUFHEAD; MSGK: USERIF.MSGKIND;  
BEGIN  
  LOOP  
    USERIF.MSGIN(M,MSGK);  
    VT.FROMUSER(M,MSGK);  
  END; (*LOOP*)  
END USERTOVT;  
  
PROCEDURE VTTOUSER;  
(*DATA FLOW FROM THE VIRTUAL TERMINAL TO THE USER*)  
VAR M: BUFHEAD; MSGK: USERIF.MSGKIND;  
BEGIN  
  LOOP  
    VT.TOUSER(M,MSGK);  
    USERIF.MSGOUT(M,MSGK);  
  END; (*LOOP*)  
END VTTOUSER;  
  
PROCEDURE TSIT;  
(*INTERRUPT FROM THE TRANSPORT SERVICE TO THE VIRTUAL  
  TERMINAL*)  
VAR IT: BUFHEAD;  
BEGIN  
  LOOP  
    TSIF.ITIN(IT);  
    VT.ITFROMTS(IT);  
  END; (*LOOP*)  
END TSIT;  
  
PROCEDURE TSTOVT;  
(*DATA FLOW FROM THE TRANSPORT SERVICE TO THE VIRTUAL  
  TERMINAL*)  
VAR L: BUFHEAD;  
BEGIN  
  LOOP  
    TSIF.LETTERIN(L);  
    VT.FROMTS(L);
```



```
    END; (*LOOP*)  
END TSTOVT;
```

```
PROCEDURE VTIT;  
(*INTERRUPT FROM THE VIRTUAL TERMINAL TO THE TRANSPORT  
SERVICE*)  
VAR IT: BUFHEAD;  
BEGIN  
    LOOP  
        VT.ITTOTS(IT);  
        TSIF.ITOUT(IT);  
    END; (*LOOP*)  
END VTIT;
```

```
PROCEDURE VTTOTS;  
(*DATA FLOW FROM THE VIRTUAL TERMINAL TO THE TRANSPORT  
SERVICE*)  
VAR L: BUFHEAD;  
BEGIN  
    LOOP  
        VT.TOTS(L);  
        TSIF.LETTEROUT(L);  
    END; (*LOOP*)  
END VTTOTS;
```

```
BEGIN  
    STARTPROCESS(USERTOVT,ADR(WSP1),TSIZE(WSP));  
    STARTPROCESS(VTTOUSER,ADR(WSP2),TSIZE(WSP));  
    STARTPROCESS(TSIT,ADR(WSP3),TSIZE(WSP));  
    STARTPROCESS(TSTOVT,ADR(WSP4),TSIZE(WSP));  
    STARTPROCESS(VTIT,ADR(WSP5),TSIZE(WSP));  
    VTTOTS;
```

```
END VTPRCS.
```



We have allocated the storage for the procedures to be started as processes statically /WSPi/, but we could have done it also dinamically, with the help of the NEW statement. The last process /VTTOTS/ is called like a procedure, because the Modula-2 starting system should be regarded as a process as well.

Beside the main components there are still several others, like:

- Command Interpreter;
- Storage Management;
- Buffer management;
- Queueing;
- Process scheduling.

All these components have been realized as definition/implementation module pairs. If we are successful in writing the definition module, we may undertake later any kind of changes in the implementation modules, without changing anything else. During our development we had to change sometimes some definition modules as well, then we had to recompile several modules. It is somewhat unpleasent, but it forces the programmer to be as careful as possible with the definition of his interfaces. This is a great advantage indeed!

#### Queueing and synchronising in the Virtual Terminal Model.

Modula-2 does not contain an integrated process-scheduler /as mentioned already/, consequently, it cannot check the proper use of the synchronising primitives. /That is the price for the freedom in writing own process-schedulers./ Thus, the Modula-2 programmer must be careful with process synchronising and must keep himself to strict rules, if he wants to have a reliable system.



Our basic tool for process synchronisation is the interface module, as described in /1/. Its concept is based on the monitor concept of Brinch-Hansen and Hoare /8/. The interface module assures the mutual exclusion on its procedures. As Modula-2 regards processes as coroutines, the implementation of an interface module is trivial, i.e. it stands for every module, that if one process enters it, no other process may enter it, until the first process leaves it /the WAIT and SEND operations imply also leaving/. The synchronisation primitives used, are offered by the process-scheduler /WAIT, SEND etc./. A strong discipline must be followed in using them. Corresponding WAIT-SEND/DOWN/ pairs must stay in the same module.

The synchronisation of the processes of the Virtual Terminal Model are concentrated into one single module, namely into the VTQ module. The VTQ module contains all queues of the VT module, i.e. they contain data, that are already processed by the VT procedures, and are to leave the VT module in the direction of the user or the network. Incoming data will not be queued, but will be processed immediately. We show the entire VTQ module /in a little simplified form, as the real model handles also priorities in VTQ/.

```
DEFINITION MODULE VTQ;

FROM SYSTEM IMPORT WORD;
FROM FIFOQ IMPORT QUEUE;

EXPORT QUALIFIED ENQ, DEQ, QKIND;

TYPE QKIND = (MSGQ, USERQ, TSITQ, TSQ);

PROCEDURE ENQ(Q: QKIND; THIS: WORD);
(*ENQUES THIS IF POSSIBLE. IF NOT, IT WAITS*)

PROCEDURE DEQ(Q: QKIND; VAR THIS: WORD);
(*DEQUEUES THE NEXT ELEM IF POSSIBLE. IF NOT, IT
  WAITS*)

END VTQ.
```



The definition module imports two types; the type WORD from the module SYSTEM /the queue elements may be words/ and the type QUEUE from the module FIFOQ, that realizes a first-in/first-out based queue handling. The VTQ module exports the type QKIND, that describes the kind of queues available in the model, and two operations on it: ENQ for enqueueing and DEQ for dequeueing. The way of operation of these procedures is unknown at this point, the only thing we must know that they will be completed in a finite time.

The implementation module of VTQ shows the realization of the above operations:

```
IMPLEMENTATION MODULE VTQ;

FROM SYSTEM IMPORT WORD;
FROM PROCESSSCHEDULER IMPORT
  INIT SIGNAL, SIGNAL, WAIT, SEND;
FROM FIFOQ IMPORT CREATED, FULLQ, EMPTYQ, GETQ, PUTQ, QUEUE;

CONST QLENGTH = 3; (*ALL QUEUES HAVE THE SAME LENGTH*)

VAR NONFULL, NONEMPTY : ARRAY QKIND OF SIGNAL;
    VTQUEUES: ARRAY QKIND OF QUEUE;
    I: QKIND;

PROCEDURE ENQ(Q: QKIND; THIS: WORD);
  (*ENQUES THIS IF POSSIBLE. IF NOT, IT WAITS*)
  BEGIN
    IF FULLQ(VTQUEUES[I]) THEN WAIT(NONFULL[I]) END;
    PUTQ(VTQUEUES[I], THIS);
    SEND(NONEMPTY[I]);
  END ENQ;
```

```
PROCEDURE DEQ(Q: OKIN); VAR THIS: WORD;  
(*DEQUEUES THE NEXT ELEM IF POSSIBLE. IF NOT, IT  
  WAITS*)  
BEGIN  
  IF EMPTYQ(VTQUEUESEQ) THEN WAIT(NONEMPTYEQ) END;  
  GETQ(VTQUEUESEQ, THIS);  
  SEND(NONFULLEQ);  
END DEQ;  
  
BEGIN  
  FOR I:= MSGQ TO TSG DO  
    INITIGNAL(NONFULLI); INITIGNAL(NONEMPTYI);  
    CREATED(VTQUEUESCI, LENGTH);  
  END; (*FOR*)  
  
END VTQ.
```

The implementation module imports from the module PROCESSSSCHEDULER the type SIGNAL and the necessary operations on it, that have been described earlier. It imports from FIFOQ the type QUEUE and the corresponding operations: CREATQ /creates a queue with given length/, FULLQ, EMPTYQ, /Boolean functions, give true if the queue is full, respectively empty/, PUTQ /enqueuees an element into the queue/ GETQ /dequeues an element from the queue/.

The implementation module contains the array variable VTQUEUES; one queue type element for each kind of queue. It is an essential point, that VTQUEUES is hidden from the environment /as it is not exported/.

Thus, the queues of the model are protected from any kind of use, except via ENQ and DEQ. The array variables NONFULL and NONEMPTY serve for the synchronisation of the processes. If a request arrives for taking out an element from an empty queue, respectively to put in an element into a full queue, then the requesting process starts to wait on the corresponding element of the signal array NONEMPTY, respectively NONFULL. The appropriate signal element will be activated /by SEND/ whenever an action has been taken on the queue.



## CONCLUSIONS

We have used the Modula-2 language to develop a Virtual Terminal Model. We have built also a stand-alone system to assist the model. This system contains facilities like process-scheduling, queueing, buffering etc. The work has been done surprisingly quickly /three man-months/, and with few errors. Most bugs could be found simply by inspecting the source listings. After having corrected some bugs in the process-scheduler, we had not one single error because of scheduling.

Modula-2 has not only supported the task of decomposing, but it also influenced the design in choosing good structures. The definition/implementation module concept is extremely attractive and convenient for thinking in modules. The only drawback, that we have encountered, is the fact, that modules may not be multiplied. We understand, that it is not their purpose either, as they are just syntactical "walls" for information hiding. As a matter of fact, we had rather much trouble for the lack of any possibility to express several instances of abstract data types in an easy manner.

Modula-2 has also encouraged us to choose a scheduler of our own taste, and to invite concurrency into the structure without difficulty.

Modula-2 has proved to be an excellent tool for the implementation of a complex task, and also to be a good specification language. It was pleasant to work with it, and we think, it is the greatest advantage of the project, that we can trust our own program, much more than generally.



## ACKNOWLEDGEMENTS

I express my thanks to prof. N.Wirth, L.Geissmann and Ch. Jacobi for making me the Modula-2 compiler available and for giving many helpful ideas.

I thank Ann Dünki for many excellent ideas.

I also thank my colleagues L.Almási, L.Csaba, M.Papp, I. Tétényi and M.Szabó for many valuable helps and remarks.

My most sincere thanks to A.Ercsényi, who has actually been a co-author in the entire project, and who gave the better part of the ideas.

## REFERENCES

1. N.Wirth, "Modula-2", ETH-Berichte 36, 1980.
2. N.Wirth, "Modula: A language for modular multiprogramming" SOFTWARE - Practice and Experience 7, No.1.3-35 /1977/
3. N.Wirth, "Design and Implementation of Modula" SOFTWARE-Practice and Experience, 7, No.1.67-84 /1977/
4. K.V.Le, "The module: a tool for structured programming" ETH - Diss: 6153, 1978
5. L.Geissman, "Modulkonzept und separate Compilation in der Programmiersprache Modula-2" In W.Remmele, H.Schecher, Eds., "Mikrocomputing", 98-114, Stuttgart, 1979.
6. P.Brinch Hansen "The architecture of concurrent programs" Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1977.
7. J.Hoppe "A simple nucleus written in Modula-2" ETH - Berichte 35, 1980.
8. C.A.R. Hoare "Monitors: an operating system structuring concept" STAN-CS-73-401, November 1973.



9. A.Dünki, P.Schicker "Page Virtual Terminal Formal Implementation in Pascal" EIN/ZHR/77/024 1977.
10. T.Joachim, "Implementation du protocole standard X.25 à partir d'un modèle de formalisation et de mécanismes abstraits de programmation" Document de travail 103, Université de Montreal, 1977.
11. G.Bochmann and T.Joachim "Development and structure of an X.25 implementation" Université de Montreal, April 1978.
12. "Proposal for a standard Virtual Terminal Protocol" INWG PROTOCOL, February 1978.



## INDUCTIVE DOMAINS AND ALGEBRAIC SEMANTICS OF CF LANGUAGES\*

Stephen D. Comer\*

Mathematical Institute, Oxford  
Great – Britain

### 1. Introduction

This note contains two simple observations on the effect of allowing a CF language to admit inductive definitions. Such languages can be generated by an unambiguous grammar and they allow the construction of an "adequate" algebraic semantics in the sense of Andreka – Nemeti – Sain [2] (henceforth referred to as ANS).

The development of an algebraic semantics for a well presented language  $L = \langle S, M, k \rangle$  in [2] depends on using a grammar  $G$  that satisfies an "adequateness criterion". Section 3 of [2] presents examples of grammars that are adequate and those that are not. A careful analysis of these examples reveals a common thread. In all cases the "meaning function"  $k$  for  $L$  was defined by induction on the complexity of the words in  $S$ . This, of course, is the most common way of defining a function on  $S$ . In most cases where  $G$  was not adequate for  $L$  it happens that  $G$  is an ambiguous grammar and it is precisely the ambiguity that leads to the inadequacy. In a nutshell, the rewrite rules of  $G$  conflict with the inductive clauses used to define  $k$ . The point is that the ability to make inductive definitions on the syntax  $S$  of a language implicitly gives a "parse" of  $S$ . We formalize this below (Proposition 1) and show that, in the situation where the meaning function is defined by induction, the induced unambiguous grammar is always adequate (Proposition 2).

### 2. Preliminaries

We briefly review terminology used in ANS [2].

A *well presented language* is a triple  $L = \langle S, M, k \rangle$  where  $S$  is a nonempty set (the *syntax* of  $L$ ) defined by a generative grammar  $G$ ,  $M$  is a nonempty set-theoretically defined class (the *models* of  $L$ ), and  $k$  is a function on  $S \times M$  (the *meaning function* of  $L$ ) that is also assumed to be set-theoretically defined.

We assume throughout that the grammar  $G$  generating  $S$  is context-free (CF). Formally,  $G = \langle N, X, \langle R_i : i \in I \rangle \rangle$  with nonterminals  $N$ , terminal symbols

\* This work was supported by National Science Foundation Grant MCS-8003896.  
Author's address after July 1981: The Citadel, Charleston, S.C. 29409, USA.



$X(N \cap X = \emptyset)$ , and rewrite rules (or productions)  $R_i, i \in I$ . Each  $R_i$  has the form  $A \vdash u$  for some  $A \in N$  and  $u \in (N \cup X)^*$ . For each  $A \in N$  the *syntactic category of A* is the set

$$S_A = \{u \in X^* : A \vdash * u\}.$$

The syntax  $L(G)$  generated by  $G$  is defined as  $L(G) = \cup \{S_A \mid A \in N\}$ , which we assume equals  $S$ .

The first task in developing an algebraic semantics for  $L$  is to turn  $S$  into a "syntactic algebra", actually an  $N$ -sorted algebra or operator domain  $\underline{S}$  (see ADJ[1] or ANS[2]). The universe assigned to the sort  $A \in N$  is  $S_A$ . The operations on  $\underline{S}$  are derived from the production rules in the following way. If  $R_i$  is the rule

$$A_0 \vdash u_0 A_1 u_1 \dots A_n u_n$$

where  $u_j \in X^*$  and  $A_j \in N$  for  $j < n + 1$ , the associated operation  $F_i$  has type  $S_{A_1} \times \dots \times S_{A_n} \rightarrow S_{A_0}$  and is defined as

$$F_i(a_1, \dots, a_n) = u_0 a_1 u_1 \dots a_n u_n$$

for each  $n$ -tuple  $(a_1, \dots, a_n) \in S_{A_1} \times \dots \times S_{A_n}$ .

For each  $\nu \in M$ ,  $k(-, \nu) = \lambda u. k(u, \nu)$  is a function on  $S$ , hence a function on the algebra  $\underline{S}$ . A grammar  $G$  is called *adequate* for  $L$  (see ANS[2]) if  $k(-, \nu)$  is a homomorphism on  $\underline{S}$  for each  $\nu \in M$ .

We also use the following terminology about sequences. Suppose  $\sigma, \tau \in S \subseteq X^*$ . We say  $\sigma$  is a *part* of  $\tau$ , in symbols  $\sigma \sqsubseteq \tau$ , if  $\tau = \mu \sigma \nu$  for some  $\mu, \nu \in X^*$ .  $\sigma$  is a *proper part* of  $\tau$  if  $\sigma \sqsubseteq \tau$  and  $\sigma \neq \tau$ .  $\sigma$  is a *maximal part* of  $\tau$  if  $\tau$  covers  $\sigma$  in the poset  $\langle S, \sqsubseteq \rangle$ .  $\tau \in S$  is called a *basic word* (or *atom*) of  $S$  if it is a minimal element in the poset  $\langle S, \sqsubseteq \rangle$ .

### 3. Inductive domains

What is going on when a function  $k$  is defined on a syntax  $S = L(G)$  by induction on the "complexity" of words? To begin with, the value of  $k$  is specified on "atoms" of  $L(G)$ . Then, for every  $\sigma \in L(G)$ , not an "atom", the value  $k(\sigma)$  is given by a function applied to values  $k(\sigma_1), \dots, k(\sigma_n)$  where  $\sigma_1, \dots, \sigma_n$  are "less complex" parts of  $\sigma$ . If  $k$  is really a function, that is, well-defined, the parts  $\sigma_1, \dots, \sigma_n$  must be uniquely determined from  $\sigma$ . Thus, for inductive definitions to be possible, a "structuring" of  $S$  must be present. This motivates the following notion of an inductive domain.

**Definition 1.** An *inductive domain* is a triple  $\langle S, \langle C_j : j \in J \rangle, j_0 \rangle$  where  $S$  is a CF syntax,  $\{C_j : j \in J\}$  is a partition of  $S$ , and  $j_0 \in J$  such that the following hold:



- (i)  $\sigma \in C_{j_0}$  iff  $\sigma$  is a basic word of  $S$ ;
- (ii) for every  $j \in J - \{j_0\}$  there is a sort  $A_0 \in N$  with  $C_j \subseteq S_{A_0}$  and there exist a unique sequence  $\langle A_1, \dots, A_k \rangle$  of sorts and unique  $\mu_i \in X^*$  (for  $i \leq k$ ) such that every  $\sigma \in C_j$  has the form

$$(\dagger) \quad \sigma = \mu_0 \tau_1 \mu_1 \dots \tau_k \mu_k$$

where  $\tau_i \in S_{A_i}$  for  $i = 1, \dots, k$  and  $\tau_1, \dots, \tau_k$  are maximal parts of  $\sigma$ . Conversely, for every choice  $\tau_i \in S_{A_i}$  for  $i = 1, \dots, k$ , the  $\sigma$  with description  $(\dagger)$  is an element of  $C_j$ .

The subsets  $C_j$  are called *clauses*. Each clause has a type.  $C_{j_0}$  has type 0. A clause  $C_j$ ,  $j \neq j_0$  has type  $k$  where  $k$  is the length of the sequence  $\langle A_1, \dots, A_k \rangle$  associated with it. We say that a syntax  $S$  admits *inductive definitions* if there exist  $\langle C_j : j \in J \rangle$  and  $j_0 \in J$  such that  $\langle S, \langle C_j : j \in J \rangle, j_0 \rangle$  is an inductive domain.  $\square$

**Definition 2.** Suppose  $\langle S, \langle C_j : j \in J \rangle, j_0 \rangle$  is an inductive domain.

- (i) An *inductive definition* over  $S$  is a system of functions  $\langle \mathbf{O}_j : j \in J \rangle$  such that domain  $\mathbf{O}_{j_0} = C_{j_0}$  and  $\mathbf{O}_j$  has type  $k$  whenever  $C_j$  has type  $k$  for all  $j \in J - \{j_0\}$ .
- (ii) A function  $h$  on  $S$  is *defined by induction* by the inductive definition  $\langle \mathbf{O}_j : j \in J \rangle$  if

$$(a) \quad h(\sigma) = \mathbf{O}_{j_0}(\sigma) \text{ for all } \sigma \in C_{j_0};$$

$$(b) \quad \text{for all } \sigma \in C_j, j \neq j_0, h(\sigma) = \mathbf{O}_j(h(\tau_1), \dots, h(\tau_k)) \text{ where}$$

$\sigma$  has the form  $(\dagger)$  in Definition 1.  $\square$

Our first goal is to characterize when a CF syntax admits inductive definitions.

**Proposition 1.** A context-free syntax  $S$  admits inductive definitions iff  $S = L(G')$  for some unambiguous grammar  $G'$  (that is, no word possesses two distinct  $G'$ -derivation trees, cf., Clark - Cowell [3]).

**Proof.** Suppose  $\langle S, \langle C_j : j \in J \rangle, j_0 \rangle$  is an inductive domain where  $S = L(G)$  for some CF grammar  $G$ . Let  $K = C_{j_0} \cup (J - \{j_0\})$ . We define  $G' = \langle N, X, \langle P_j : j \in K \rangle \rangle$  where  $N$  and  $X$  are the nonterminals and terminals, respectively, of  $G$ . The rewrite rules  $P_j, j \in K$ , of  $G'$  are as follows:



for  $\sigma \in C_{j_0} \cap S_A$ , let  $P_\sigma$  be  $A \vdash_{G'} \sigma$ ;

for each clause  $C_j$ ,  $j \neq j_0$  let  $P_j$  be  $A_0 \vdash_{G'} \mu_0 A_1 \mu_1 \dots A_k \mu_k$

where  $A_0, A_1, \dots, A_k$  are the unique sorts and  $\mu_0, \dots, \mu_k$  the unique sequences in (ii) of Definition 1.

An easy induction shows that  $G'$  generates  $S$ . We show that  $G'$  is unambiguous by induction on word length. Clearly each basic word of  $S$  has exactly one derivation tree. Now consider a word  $\sigma$  in  $C_j$  where  $\sigma$  has the form  $(\dagger)$  in (ii) of Definition 1 and each maximal part  $\tau_i$  of  $\sigma$  has exactly one derivation tree. Any derivation tree for  $\sigma$  has root labeled  $A_0$  (recall  $C_j \subseteq S_{A_0}$ ) and the labels, left-to-right, of the sons of  $A_0$  are exactly the symbols in the sequence  $\mu_0 A_1 \mu_1 \dots A_k \mu_k$  and the subtree rooted at each  $A_i$  is exactly the derivation tree for  $\tau_i$ . By the uniqueness in (ii) of Definition 1 and the induction hypothesis,  $\sigma$  has exactly one derivation tree.

For the converse, suppose,  $S = L(G)$  where  $G$  is an unambiguous CF grammar  $\langle N, X, \langle R_i : i \in I \rangle \rangle$ . Let  $I' = \{i \in I : R_i \text{ has the form } A \vdash_G \sigma \text{ for some } A \in N \text{ and } \sigma \in X^*\}$ .  $I' \neq \emptyset$  (we disallow  $A \vdash A$  as a production) so fix  $i_0 \in I'$  and define clause  $C_{i_0} = \{\sigma \in X^* : A \vdash_G \sigma \text{ for some } A \in N\}$ . For each  $i \in I - I'$ , the rewrite rule  $R_i$  contains nonterminals on the right-hand side. Suppose  $R_i$  has the form

$$A_0 \vdash_G \mu_0 A_1 \dots A_k \mu_k$$

for some  $\mu_i \in X^*$  and  $A_i \in N$  ( $i \leq k$ ). Define the clause  $C_i$  as

$$C_i = \{\sigma \in X^* : \sigma = \mu_0 \tau_1 \mu_1 \dots \tau_k \mu_k \text{ for some } \tau_1 \in S_{A_1}, \dots, \tau_k \in S_{A_k}\}.$$

Using the fact  $G$  is unambiguous, it is easily checked that  $\langle S, \langle C_j : j \in (I - I') \cup \{i_0\} \rangle, i_0 \rangle$  is an inductive domain.  $\square$

For a CF syntax  $S$  which admits inductive definitions we call the grammar  $G'$  for  $S$  constructed in Proposition 1 the *grammar induced by the inductive structure*. It is easy to give examples of CF languages that can be generated by different unambiguous grammars; see ANS, Example 3(i).

We now consider the relationship between having an inductively defined meaning function and an adequate grammar.

**Proposition 2.** Suppose  $L = \langle S, M, k \rangle$  is a well presented language. Then the existence of an inductive definition for  $k(-, \mathcal{M})$ , for each  $\mathcal{M} \in M$ , is equivalent to the existence of an adequate grammar for  $L$ .



**Proof.** First, suppose  $S$  admits inductive definitions and, for each  $\mathcal{M} \in M$ ,  $k(-, \mathcal{M})$  is defined by induction. We claim the grammar for  $S$  induced by the inductive structure is adequate for  $L$ . It needs to be shown, for  $\mathcal{M} \in M$ , that the equivalence relation  $\equiv_{\mathcal{M}}$  on  $\underline{S}$ , defined by  $\sigma \equiv_{\mathcal{M}} \tau$  iff  $k(\sigma, \mathcal{M}) = k(\tau, \mathcal{M})$ , is a congruence relation on  $\underline{S}$ . Towards this end, suppose  $F_j$  is a  $k$ -ary operation of the type  $S_{A_1} \times \dots \times S_{A_k} \rightarrow S_{A_0}$  associated with a clause  $C_j$ ,  $j \neq j_0$ , and suppose  $\sigma_i \equiv_{\mathcal{M}} \tau_i$  for all  $i = 1, \dots, k$ ,  $\sigma = F_j(\sigma_1, \dots, \sigma_k)$  and  $\tau = F_j(\tau_1, \dots, \tau_k)$ . Then, by Definition 2(ii),

$$\begin{aligned} k(\sigma, \mathcal{M}) &= O_j(k(\sigma_1, \mathcal{M}), \dots, k(\sigma_k, \mathcal{M})) \\ &= O_j(k(\tau_1, \mathcal{M}), \dots, k(\tau_k, \mathcal{M})) \\ &= k(\tau, \mathcal{M}), \end{aligned}$$

hence  $\sigma \equiv_{\mathcal{M}} \tau$  as desired.

Conversely, if  $G$  is adequate for  $L$ , the homomorphism property for  $k(-, \mathcal{M})$  on  $\underline{S}$  gives an inductive definition of the function  $k(-, \mathcal{M})$  on  $S$ .  $\square$

#### 4. Example

We illustrate the remarks in the preceeding section for the implication language  $L = \langle S, M, k \rangle$  treated in ANS[2], Example 3(iii), Section 3.  $S$  is generated by the grammar  $G_6 = \langle N, X, \langle R_i : i \in I_6 \rangle \rangle$  where  $N = \{F\}$ ,  $X = \{p_i : i \in \omega\} \cup \{\rightarrow\}$ ,  $I_6 = \omega \cup \{\rightarrow\}$  and the productions are:

$$\begin{aligned} R_{\rightarrow} : F &\vdash F \rightarrow F \\ R_i : F &\vdash p_i \quad (\text{for } i \in \omega). \end{aligned}$$

The class of models is  $M = {}^\omega 2$ . The meaning function  $k$  is defined, for  $\mathcal{M} \in {}^\omega 2$ , by induction as

$$\begin{aligned} k(p_i, \mathcal{M}) &= \mathcal{M}(i) \\ k(p_i \rightarrow \phi, \mathcal{M}) &= \begin{cases} 0 & \text{if } \mathcal{M}(i) = 1 \text{ and } k(\phi, \mathcal{M}) = 0 \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

for each  $i \in \omega$ . The grammar  $G_6$  is not adequate for  $L$ . An adequate grammar can be obtained from  $G_6$  in either of two ways, depending upon what we desire to preserve.



**Method 1.** Keep the meaning function.

We must change the grammar. The inductive definition of  $k$  implicitly gives  $S$  the structure of an inductive domain. The induced grammar has rewrite rules, for  $i \in \omega$ ,

$$F \vdash p_i \rightarrow F$$

$$F \vdash p_i.$$

This is exactly the grammar  $G_7$  considered in ANS[2] and is adequate by the proof of Proposition 2.

**Method 2.** Keep the grammar (but make unambiguous by parsing). In the case of  $G_6$ , extend  $X$  to  $X' = X \cup \{ (, ) \}$  and let

$$R'_\rightarrow : F \vdash (F \rightarrow F)$$

$$R'_i : F \vdash p_i \quad (\text{for } i \in \omega).$$

Then  $G' = \langle N, X', \langle R'_i : i \in I_6 \rangle \rangle$  is an unambiguous grammar which will admit inductive definitions. In particular, we can define a new meaning function  $k'(-, \omega_2)$  for  $\omega_2$ , by

$$k'(p_i, \omega_2) = \omega_2(i), \text{ for each } i \in \omega, \text{ and}$$

$$k'((\phi \rightarrow \psi), \omega_2) = \begin{cases} 0 & \text{if } k'(\phi, \omega_2) = 1 \text{ and } k'(\psi, \omega_2) = 0 \\ 1 & \text{otherwise.} \end{cases}$$

$G'$  is the grammar induced by the inductive structure, so by Proposition 2,  $G'$  is adequate for the language  $\langle S', M, k' \rangle$ . Of course,  $S' = L(G')$  is slightly different from  $S$  due to the added punctuation.

The other examples of inadequate grammars in ANS[2] can be modified in the same way.

## 5. Concluding remarks

Semantics (or meaning functions) used in the vast majority of languages are defined by induction. Thus, for all practical purposes, in dealing with context-free languages, we may as well assume we have such a language. The content of Proposition 2 shows that, under this assumption, an adequate algebraic semantics can always be constructed.

The content of Proposition 1 may be viewed as a justification for assuming from the beginning the syntax is "parsed" (as in, for example, ADJ[1]). The inductive structure induces a natural unambiguous grammar that generates the syntax anyway. On this point we quote Clark and Cowell [3], page 159.



"When a context-free grammar is used to specify the syntax of a programming language it is clearly important that the grammar be unambiguous. For, since the productions used in the generation of the program indicate the way the program should be 'parsed' and its meaning derived, the existence of two distinct parses for the same program might lead to an interpretation of the program by the compiler different from the interpretation intended by the programmer."

The author is grateful to H. Andréka and I. Németi for their encouragement and correspondence concerning the ideas presented in this paper.

### References

- [1] ADJ. Gougen, J.A., Thatcher, J.W., Wagner, E.G., and Wright, J.B., Initial Algebra Semantics and Continuous Algebras. JACM 24(1977), 68-95.
- [2] ANS. Andréka, H., Németi, I., and Sain, I., Connections between Algebraic Logic and Initial Algebra Semantics of CF languages Part I and Part II. Preprint, Math. Inst. Hung. Acad. Sci. October 1978. Appeared in: Mathematical Logic in Computer Science (Proc. Coll. held in Salgótarján 1978), Dömölki, B., Gergely, T. (Eds), Colloq. Math. Soc. J. Bolyai Vol. 26, North-Holland, 1981. Part I: pp. 25-83, Part II: pp. 561-606.
- [3] Clark, K.L. and Cowell, D.F., Programs, Machines, and Computation, McGraw-Hill, 1976.









## SOME BASIC PROPERTIES OF $k$ -BOUNDED INTERPRETATIONS OF GRAMMAR FORMS

*Erzsébet Csuhaj Varjú*

Computer and Automation Institute  
Hungarian Academy of Sciences  
Budapest, Hungary

### 1.

#### INTRODUCTION

The notion of a grammar form has been introduced to formalize the idea of a master grammar defining a family  $G(F)$  of structurally related grammars. The grammars in the associated family are obtained by an interpretation mechanism, which involves a substitution of the nonterminals and terminals by disjoint sets of nonterminals and sets of terminal words, respectively. Therefore  $G$  may be regarded as an operator which when applied to a grammar yields a set of grammars.

From the mathematical point of view it is very natural to ask what happens when some restrictions are made on the interpretation mechanism. In this paper a variant  $G_k$  of  $G$ , called the  $k$ -bounded interpretation operator, is investigated which also yields a set of grammars when applied to a given grammar form.

The grammar  $F' = (V', S'; P'; \sigma')$  is said to be a  $k$ -bounded interpretation grammar of a grammar form  $F = (V, S, P, \sigma)$ , if  $F'$  is an element of its grammar family and the minimal number of new nonterminals which are necessary to create  $F'$  from  $F$  is at least  $k$  for each nonterminal  $\xi$  of  $F$  occurring in  $P$ . The set of all  $k$ -bounded interpretation grammars, for all  $j \leq k$ , of a grammar form is called its  $k$ -bounded grammar family. These notions came from S.Ginsburg and G.Révész [4].



Organizationally, there are four sections in the paper. In section 2 some preliminary notions are reviewed.

In section 3 the notion of a  $k$ -bounded interpretation and that of a  $k$ -bounded interpretation grammar of a grammar form are defined. We define the  $k$ -bounded interpretation operator, too. We introduce two new notions, namely the notion of a  $k$ -th inflation and that of a  $k$ -th inflation grammar of a grammar form, which serve as useful tools in the characterization of  $k$ -bounded grammar families. These notions express maximum properties. Then some properties of the  $k$ -bounded interpretation operator are proved. Particularly, we show that set of  $k$ -th inflation operators  $/k=1,2,3,.../$  is a monoid and the set of  $k$ -bounded interpretation operators  $/k=0,1,2,.../$  is a semiring. It is shown that the  $k$ -bounded grammar family of a grammar form is equal to the union of  $1$ -bounded grammar families of its  $k$ -th inflation grammars.

In section 4 the notion of pseudo-isomorphism is reviewed. We introduce the notion of the skeleton grammar of a grammar form which is a simple modification of the notion of an interpretation grammar completed with respect to terminals of a grammar form. We define the corresponding operators, too. After this we characterize the relation of the  $k$ -bounded interpretation operator, the  $k$ -th inflation operator and these two operators. We show that the pseudo-isomorphism operator commute with each other. Then it is presented that the  $k$ -th inflation operator and the skeleton operator commute with each other, too.



2.

PRELIMINARIES

We first recall some elementary ideas of grammar forms.

A grammar  $G=(V,S,P,\sigma)$  is a quadruple, where  $V$  is a finite nonempty set of symbols,  $S$  is a nonempty set of terminals,  $V-S$  is a nonempty set of nonterminals,  $\sigma$  in  $V-S$  is the startsymbol, and  $P \subseteq \{\alpha \rightarrow \beta \mid \alpha \in V^+, \beta \in S^*\}$  is a finite nonempty set of rules.

A symbol occurring in at least one production of a grammar is called an inner symbol of it. A symbol of a grammar, which is not an inner symbol of it, is said to be an outcast symbol of it. We use the term of an inner terminal /nonterminal/ and that of an outcast terminal /nonterminal/ in the respective way.

Those grammars which have not at least one inner terminal are said to be  $t$ -empty.

A  $t$ -empty grammar is said to be symbol-tight if its each nonterminal which is different from the startsymbol is an inner symbol of it. A grammar which is not  $t$ -empty is said to be symbol-tight if each symbol of it which is different from the startsymbol is an inner symbol of it.

In the following let  $V$  be a fixed infinite set of abstract symbols and  $\Sigma$  a subset of  $V$  such that  $\Sigma$  and  $V-\Sigma$  are both infinite.

DEFINITION 2.1

A quadruple  $F=(V,S,P,\sigma)$  is called a grammar form if  $(V,S,P,\sigma)$  is a grammar and  $V-S \subseteq V-\Sigma$ ,  $S \subseteq \Sigma$  hold.

The set of all grammar forms is denoted by  $\mathcal{F}$ .

The notions, which are defined previously, are applied to grammar forms in the stated way.

A grammar form  $F'=(V',S',P',\sigma)$  is called a subform of a grammar form  $F=(V,S,P,\sigma)$  if  $P' \subseteq P$ ,  $S' \subseteq S$ ,  $V'-S' \subseteq V-S$  hold.



# NOTATION

The mapping  $SYM$  from  $\mathcal{F}$  into  $2^{\mathcal{F}}$  is defined as follows:  
For each grammar form  $F$  let  $SYM(F) = \{F'\}$ , where  $F'$  is the maximal symbol-tight subform of  $F$ . For each set  $\mathcal{F}_1$  of grammar forms let  $SYM(\mathcal{F}_1) = \bigcup_{F \in \mathcal{F}_1} SYM(F)$  and let  $SYM(\emptyset) = \emptyset$ .

The way a grammar form specifies that a grammar form is "structurally close" to it is accomplished by the next fundamental notion:

## DEFINITION 2.2

The 5-tuple  $I = (\mu, V', S', P', \sigma')$  is said to be an interpretation of the grammar form  $F = (V, S, P, \sigma)$ , if  $(V', S', P', \sigma')$ -denoted by  $F'$ -is a grammar form and  $\mu$  is a finite substitution on  $V^*$  such that the following conditions hold:

- /i/  $\mu(a) \subseteq \Sigma^*$  for each terminal  $a$  of  $F$ ;
- /ii/  $\mu(\xi) \subseteq V - \Sigma$  for each nonterminal  $\xi$  of  $F$ ;
- /iii/  $\mu(\xi) \cap \mu(\eta) = \emptyset$  for all different nonterminals  $\xi$  and  $\eta$  of  $F$ ;
- /iv/ each nonterminal of  $F'$  is an element of  $\mu(V - S)$  and each terminal of  $F'$  occurs in at least one element of  $\mu(S)$ ;
- /v/  $P' \subseteq \mu(P)$ , where  $\mu(P) = \{u \rightarrow v \mid u \in \mu(\alpha), v \in \mu(\beta) \text{ and } \alpha \rightarrow \beta \in P\}$ ;
- /vi/  $\sigma' \in \mu(\sigma)$ .

The grammar form  $F'$  is called an interpretation grammar of  $F$  and  $\mu$  is called an interpretation substitution of it.

Next we present the notion of the interpretation operator.

## DEFINITION 2.3

The mapping  $G$  from  $\mathcal{F}$  into  $2^{\mathcal{F}}$  is said to be the interpretation operator if for each grammar form  $F$ :

$$G(F) = \{F' \mid F' \text{ is an interpretation grammar of } F\}.$$

We extend the definition of this mapping to the subsets of  $\mathcal{F}$  in the following way: for each nonempty subset  $\mathcal{F}_1$  of  $\mathcal{F}$  let

$$G(\mathcal{F}_1) = \bigcup_{F \in \mathcal{F}_1} G(F) \text{ and let } G(\emptyset) = \emptyset.$$



The next theorem asserts that this operator is an idempotent mapping. Since the proof is a simple modification of the proof of lemma 1.1 of [1], therefore is omitted.

### THEOREM 2.1

For each grammar form  $F$   $G^2(F) = G(F)$ .

## 3.

### $k$ -BOUNDED INTERPRETATION OPERATORS

In this section the notion of a  $k$ -bounded interpretation and that of a  $k$ -th inflation of a grammar form are introduced. The corresponding interpretation grammars and operators are defined, too. Some theorems are presented concerning algebraic properties of these operators. The next concept represents the minimal number of new non-terminals per an old nonterminal which are necessary to create a given interpretation grammar from a given grammar form.

### DEFINITION 3.1

Let  $F' = (V'; S'; P'; \sigma')$  be an interpretation grammar of the grammar form  $F = (V, S, P, \sigma)$ . Let  $H$  be the set of interpretation substitutions of  $F$ . The grammar form  $F'$  is said to be a  $k$ -bounded interpretation grammar of  $F$  if the following hold:

/i/  $\min_{\mu \in H} \max_{\xi \text{ is an inner non-terminal of } F} \text{card} \{ \eta \mid \eta \in \mu(\xi), \eta \text{ is an inner symbol of } F' \} = k;$

and there is an element  $\mu$  of  $H$  satisfying this minimum property such that:

/ii/  $\text{card} \{ \sigma' \} \cup \{ \sigma'' \mid \sigma'' \in \mu(\sigma), \sigma'' \text{ is an inner symbol of } F' \} \leq k.$



An interpretation  $I=(\mu, V', S', P', \sigma')$  of a grammar form  $F=(V, S, P, \sigma)$  is said to be a  $k$ -bounded interpretation of it if  $(V', S', P', \sigma')$  - denoted by  $F'$  - is a  $k$ -bounded interpretation grammar of  $F$  and  $\mu$  satisfies the conditions /i/ and /ii/.

The set of all  $j$ -bounded interpretation grammars of a grammar form, where  $j$  is at most  $k$ , is said to be the  $k$ -bounded grammar family of it.

The next concept expresses maximality.

### DEFINITION 3.2

Let  $F'=(V', S', P', \sigma')$  given by the interpretation  $I=(\mu, V', S', P', \sigma')$  be an interpretation grammar of the grammar form  $F=(V, S, P, \sigma)$ .  $F'$  is called a  $k$ -th inflation grammar of  $F$  and  $I$  is called a  $k$ -th inflation of it if the following conditions hold:

- /i/  $\text{card} \{ \eta \mid \eta \in \mu(\xi), \eta \text{ is an inner nonterminal of } F' \} = k$  for each inner nonterminal  $\xi$  of  $F$ ;
- /ii/  $\mu(a) = \{a\}$  for each terminal  $a$  of  $F$ ;
- /iii/ if  $\sigma$  is an inner symbol of  $F$  then  $\sigma'$  is an inner symbol of  $F'$ ;
- /iv/  $P' = \bar{\mu}(P)$  where  $\bar{\mu}$  is a substitution defined on  $V^*$  such as follows: for each inner nonterminal  $\xi$  of  $F$   $\bar{\mu}(\xi)$  is the maximal subset of  $\mu(\xi)$  consisting of inner nonterminals of  $F'$  and for each other element  $x$  of  $F$   $\bar{\mu}(x) = \mu(x)$ .

### Remark 3.1

It comes with simple considerations that if  $F'=(V', S', P', \sigma')$  is a  $k$ -th inflation grammar of a grammar form  $F=(V, S, P, \sigma)$  then  $F'$  is a  $k$ -bounded interpretation grammar of it. Otherwise, if  $F'$  is given by the  $k$ -th inflation  $I=(\mu, V', S', P', \sigma')$  then there is not an element of  $(\mu(P) - P')$  such that each element of  $V - \Sigma$  occurring in it is an inner nonterminal of  $F'$ . That is,  $P'$  has a maximum property.

Now we are sufficiently prepared to give our definition of the corresponding mappings.



### DEFINITION 3.3

The mappings  $G_k$  and  $INF_k$  /  $k$  is a positive integer/ from  $\mathcal{F}$  into  $2^{\mathcal{F}}$  are said to be the  $k$ -bounded interpretation operator and the  $k$ -th inflation operator, respectively, if for each grammar form  $F$ :

$G_k(F) = \{F' \mid F' \text{ is an element of the } k\text{-bounded grammar family of } F\}$  ; and  $INF_k(F) = \{F' \mid F' \text{ is a } k\text{-th inflation grammar of } F\}$ .

We extend this definition to the subsets of  $\mathcal{F}$  as follows: for each nonvoid subset  $\mathcal{F}_1$  of  $\mathcal{F}$  let  $G_k(\mathcal{F}_1) = \bigcup_{F \in \mathcal{F}_1} G_k(F)$ , let

$INF_k(\mathcal{F}_1) = \bigcup_{F \in \mathcal{F}_1} INF_k(F)$  and for the empty set let  $G_k(\emptyset) = \emptyset$  and

$INF_k(\emptyset) = \emptyset$ .

We next consider a theorem concerned with the commutativity of the inflation operators.

### THEOREM 3.1

For each grammar form  $F$  and for all positive integers  $k$  and  $l$

$$INF_{kl}(F) = INF_k(INF_l(F)).$$

#### Proof

Let  $F = (V, S, P, \sigma)$  be a grammar form. We first show that

$INF_{kl}(F) \subseteq INF_k(INF_l(F))$ . Let  $F' = (V', S, P', \sigma')$  given by a  $kl$ -th inflation  $I = (\mu, V', S, P', \sigma')$  be a  $kl$ -th inflation grammar of  $F$ .

We construct an  $l$ -th inflation grammar  $F'' = (V'', S, P'', \sigma'')$  of  $F$  such that  $F'$  is a  $k$ -th inflation grammar of  $F''$ . Let us define a substitution  $\mu''$  on  $V^*$  in the following way:

- /i/ let  $\mu''(a) = \{a\}$  for each terminal  $a$  of  $F$ ;
- /ii/ for each inner nonterminal  $\xi$  of  $F$  let  $\mu''(\xi)$  be the union of the following two sets:
  - /1/ that maximal subset of  $\mu(\xi)$  which does not contain inner nonterminal of  $F'$
  - /2/ and a subset of  $\mu(\xi)$  consisting of  $l$  inner nonterminals of  $F'$ ;



/iii/ if  $F$  has at least one outcast nonterminal, then for each outcast nonterminal  $\xi$  of  $F$  let  $\mu''(\xi) = \mu(\xi)$ , and, finally,

/iv/ if  $\sigma'$  is an inner symbol of  $F$  then let  $\sigma' \in \mu''(\sigma)$ .

Let us denote  $V'' = S \cup \bigcup_{\xi \in V-S} \mu(\xi)$ . Let  $P''$  be the maximal subset of  $\mu''(P)$  such that each element of  $V''-S$  which occurs in at least one element of  $P''$  is an inner nonterminal of  $F'$ . Clearly,  $I'' = (\mu'', V'', S, P'', \sigma')$  is an interpretation of  $F$ . It is obvious, that the conditions /i-iii/ of definition 3.1 are satisfied. By virtue of remark 3.1 the condition /iv/ of definition 3.1 holds, too. Consequently,  $F''$  is an  $l$ -th inflation grammar of  $F$  and  $I''$  is an  $l$ -th inflation of it. We have to prove that  $F'$  is a  $k$ -th inflation grammar of  $F''$ . Let us define a substitution  $\mu'$  on  $V''^*$  in the following way:

/i/ for each terminal  $a$  of  $F''$  let  $\mu'(a) = \{a\}$ ;

/ii/ for each inner nonterminal  $\xi$  of  $F''$  let  $\mu(\xi)$  be a subset of  $\mu(\mu''^{-1}(\xi))$  consisting of  $k$  inner nonterminals of  $F'$  such of that for all different inner nonterminals  $\xi$  and  $\zeta$  of  $F''$   $\mu'(\xi) \cap \mu'(\zeta) = \emptyset$ ;

/iii/ if  $\sigma'$  is an inner symbol of  $F''$  then let  $\sigma' \in \mu'(\sigma')$ , else let  $\mu'(\sigma') = \{\sigma'\}$ ;

/iv/ if  $F''$  is not a symbol-tight grammar form, then for each outcast nonterminal  $\xi \neq \sigma$  of  $F''$  let  $\mu'(\xi) = \{\xi\}$ .

Note that  $\mu'(V''-S) = V'-S$ . Let us denote  $\tilde{P} = \mu'(P'')$ . Let  $\tilde{F} = (V', S, \tilde{P}, \sigma')$ . By the above construction  $\tilde{F}$  is an interpretation grammar of  $F''$ . It comes with simple considerations that conditions /i-iv/ of definition 3.2 hold, thus  $\tilde{F}$  is a  $k$ -th inflation grammar of  $F''$ . We show that  $\tilde{F} = F'$ . In order to prove this, it is enough to show that  $\tilde{P} = P'$ . We first show that  $P' \subseteq \tilde{P}$ . Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V'-S / 0 \leq i \leq n /$ ,  $x_j \in S^* / 0 \leq j \leq n+2 /$ ,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary production in  $P'$ . Then there exists a production  $q = x_0 \eta_0 x_1 \dots \eta_k x_{k+1} \rightarrow x_{k+2} \eta_k \dots \eta_n x_{n+2}$  in  $P$ , where  $\eta_i \in V-S / 0 \leq i \leq n /$ , such that  $\xi_i \in \mu(\eta_i)$  holds for each  $i / 0 \leq i \leq n /$ .



This means that  $p \in \mu(q)$ . / By the definition of  $\mu''$  and that of  $\mu'$  there exists a production  $r = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  of  $F''$ , where  $\xi_i \in V'' - S$  /  $0 \leq i \leq n$  / and  $\xi_i \in \mu'(\xi_i)$  for each  $i$  /  $0 \leq i \leq n$  / such that  $r \in \mu''(q)$  holds. Thus,  $p \in \tilde{P}$ . The arbitrary choice of  $p$  in  $P'$  implies that  $P' \subseteq \tilde{P}$ . Consider the reverse inclusion. Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V'' - S$  /  $0 \leq i \leq n$  /,  $x_j \in S^*$  /  $0 \leq j \leq n+2$  /,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary production in  $\tilde{P}$ . Then there exists a production

$r = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  in  $P''$ , where  $\xi_i \in V'' - S$  /  $0 \leq i \leq n$  / such that  $p \in \mu'(r)$ , that is,  $\xi_i \in \mu'(\xi_i)$  holds for each  $i$  /  $0 \leq i \leq n$  /. Then there exists a production

$q = x_0 \eta_0 x_1 \dots \eta_k x_{k+1} \rightarrow x_{k+2} \eta_{k+1} \dots \eta_n x_{n+2}$  in  $P$ , where  $\eta_i \in V - S$  /  $0 \leq i \leq n$  /, such that  $r \in \mu''(q)$ , that is,  $\xi_i \in \mu''(\eta_i)$  holds for each  $i$  /  $0 \leq i \leq n$  /. By the definitions of  $\mu''$ ,  $\mu'$  and  $\mu$  this implies that  $p \in \mu(q)$ . Consequently,  $p \in P'$ . Thus  $P' \supseteq \tilde{P}$ . Hence  $\tilde{F} = F'$ .

Since grammar forms  $F$  and  $F'$  were arbitrary, we proved that for each grammar form  $F$   $INF_{kl}(F) \subseteq INF_k(INF_l(F))$ . We complete the proof by showing that the reverse inclusion holds. Let  $F' = (V', S, P', \sigma')$  be an  $l$ -th inflation grammar of a grammar form  $F = (V, S, P, \sigma)$ , and let  $F'' = (V'', S, P'', \sigma'')$  be a  $k$ -th inflation grammar of  $F'$ . Let  $I' = (\mu', V', S, P', \sigma')$  be the corresponding  $l$ -th inflation of  $F$  and let

$I'' = (\mu'', V'', S, P'', \sigma'')$  be the corresponding  $k$ -th inflation of  $F'$ . We show that  $\tilde{I} = (\mu'' \circ \mu', V'', S, P'', \sigma'')$  is a  $kl$ -th inflation of  $F$ , that is,  $F''$  is a  $kl$ -th inflation grammar of  $F$ .

Clearly, the conditions /i-iii/ of definition 3.2 hold. In order to prove that the condition /iv/ of definition 3.2 holds, too, we have to show that  $P''$  is the maximal subset of  $((\mu'' \circ \mu')(P))$  such that each element of  $V - \Sigma$  which occurs in at least one of  $P''$  is an inner nonterminal of  $F''$ . Let us

assume the contrary. Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  where  $\xi_i \in V'' - S$  /  $0 \leq i \leq n$  /,  $x_j \in S^*$  /  $0 \leq j \leq n+2$  /,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an element of  $((\mu'' \circ \mu')(P)) - P''$  such that each  $\xi_i$ , where  $0 \leq i \leq n$ , is an inner nonterminal of  $F''$ . Then there exists a

production  $r = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  in  $P$ , where  $\xi_i \in V - S$  /  $0 \leq i \leq n$  /, such that  $p \in ((\mu'' \circ \mu')(r))$  holds. Since each



$\xi_i, 0 \leq i \leq n$ , is an inner nonterminal of  $F''$ , therefore there exist inner nonterminals  $\eta_i, 0 \leq i \leq n$ , of  $F'$ , such that  $\xi_i \in \mu''(\eta_i)$  hold for all  $i$ , where  $0 \leq i \leq n$ . Since  $F'$  is an  $l$ -th inflation grammar of  $F$ , then  $q = x_0 \eta_0 x_1 \dots \eta_k x_{k+1} \dots x_{k+2} \eta_{k+1} \dots \eta_n x_{n+2}$  is a production of  $F'$ . Since  $F''$  is a  $k$ -th inflation grammar of  $F'$ , then  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \dots x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  is a production of  $F''$ . This is a contradiction to our assumption. Hence  $I = (\mu'' \circ \mu', V'', S, P'', \sigma'')$  is a  $kl$ -th inflation of  $F$ . Since grammar forms  $F'$  and  $F''$  were arbitrary, therefore we proved that for each grammar form  $F$   $INF_k(INF_l(F)) \subseteq INF_{kl}(F)$  holds. Hence the result.

As an immediate consequence, we get the next corollary.

### COROLLARY 3.1

The elements of  $\tilde{I} = \{INF_k \mid INF_k \text{ is a } k\text{-th inflation operator, } k \text{ is a positive integer}\}$  form, with respect to composition, on  $2^{\mathcal{F}}$  a monoid which is isomorphic to the multiplicative monoid of the natural numbers.

Next we present a characterization result concerning the set of  $k$ -th inflation grammars of a grammar form.

### THEOREM 3.2

Let  $F' = (V', S, P', \sigma')$  be a  $k$ -th inflation grammar of a grammar form  $F = (V, S, P, \sigma)$ . Then

$$INF_k(F) = INF_1(F').$$

### Proof

Applying definition 3.2 and theorem 3.1 we obtain that

$INF_1(F') \subseteq INF_k(F)$ . We show that the reverse inclusion holds.

Let  $F'' = (V'', S, P'', \sigma'')$  given by a  $k$ -th inflation

$I'' = (\mu'', V'', S, P'', \sigma'')$  be an arbitrary  $k$ -th inflation grammar of  $F$ . Assume that  $F'$  is given by a  $k$ -th inflation

$P' = (\mu', V', S, P', \sigma')$  of  $F$ . We show that  $F''$  is a 1-st inflation grammar of  $F'$ . Let us define a substitution  $\mu$  on  $V'^*$  as follows:



- /i/ for each terminal  $a$  of  $F'$  let  $\mu(a) = \{a\}$ ;
- /ii/ let  $\mu(\sigma') = \{\sigma''\}$ ;
- /iii/ for each inner nonterminal  $\xi$  of  $F'$  let  $\mu(\xi) = \{\eta\}$ ,  
where  $\eta \in \mu''(\mu^{-1}(\xi))$  and  $\eta$  is an inner nonterminal  
of  $F''$  such that for all different inner nonterminals  
 $\xi_1$  and  $\xi_2$  of  $F'$   $\mu(\xi_1) \cap \mu(\xi_2) = \emptyset$  holds;
- /iv/ if  $F'$  is not a symbol-tight grammar form and  $F''$  is a  
symbol-tight one, then for each outcast nonterminal  
 $\xi \neq \sigma'$  of  $F'$  let  $\mu(\xi) = \{\eta\}$ , where  $\eta \in (V - \Sigma) - (V'' - S - \{\sigma''\})$   
such that for all different nonterminals  $\xi_1$  and  $\xi_2$   
of  $F'$   $\mu(\xi_1) \cap \mu(\xi_2) = \emptyset$ ;
- /v/ if  $F'$  and  $F''$  are not symbol-tight grammar forms, then  
for a fixed outcast nonterminal  $\xi \neq \sigma'$  of  $F'$  let  $\mu(\xi)$   
be the set of outcast nonterminals of  $F''$  different  
from  $\sigma''$  and for each outcast nonterminal  $\eta \neq \xi$   
- if it exists - let  $\mu(\eta) = \{\zeta\}$ , where  $\zeta \in (V - \Sigma) - (V'' - S)$   
such that for all different nonterminals  $\xi_1$  and  $\xi_2$   
of  $F'$   $\mu(\xi_1) \cap \mu(\xi_2) = \emptyset$ ;
- /vi/ if  $F'$  is a symbol-tight grammar form and  $F''$  is not it,  
then let  $\mu(\sigma')$  be the union of  $\{\sigma''\}$  and the set of out-  
cast nonterminals of  $F''$ .

Let  $\tilde{P}$  be the maximal subset of  $\mu(P')$  such that each element  
of  $V - \Sigma$  which occurs in at least one element of  $\tilde{P}$  is an inner  
nonterminal of  $F''$ . Let  $\tilde{F} = (V'', S, \tilde{P}, \sigma'')$ . We show that  $\tilde{F}$  is a  
1-st inflation grammar of  $F'$  and  $F = F''$ . By the definition of  $\mu$   
we obtain that  $\mu$  is an interpretation substitution of  $F'$  and  
 $\mu$  satisfies the conditions /i-iii/ of definition 3.2.. By  
virtue of remark 3.1.  $P$  satisfies the condition /iv/ of  
definition 3.2. This implies, that  $\tilde{F}$  is a 1-st inflation  
grammar of  $F'$ . We show that  $F'' = \tilde{F}$ , that is  $\tilde{P} = P''$ . For each  
production  $p$  in  $\tilde{P}$  there exist a production  $q$  in  $P'$  and a  
production  $r$  in  $P$  such that  $p \in \mu(q)$  and  $q \in \mu'(r)$ . By the  
definition of  $\mu''$  we obtain that  $p \in \mu(\mu'(r)) \subseteq ((\mu'' \circ \mu^{-1})(\mu'(r))) =$   
 $= \mu''(r)$ , therefore  $p \in P''$ . Thus  $\tilde{P} \subseteq P''$ .

Now let us assume the reverse inclusion.



Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V'' - S$   $/0 \leq i \leq n/$ ,  $x_j \in S^*$   $/0 \leq j \leq n+2/$ ,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary production in  $P''$ . Then there exists a production  $r = x_0 \zeta_0 x_1 \dots \zeta_k x_{k+1} \rightarrow x_{k+2} \zeta_{k+1} \dots \zeta_n x_{n+2}$  in  $P$ , where  $\zeta_i \in V - S$   $/0 \leq i \leq n/$ , such that  $p \in \mu''(r)$  holds. By the definition of  $\mu$  for each nonterminal  $\xi_i$   $/0 \leq i \leq n/$  which occurs in  $p$ , holds that there exists an inner nonterminal  $\eta_i$  of  $F'$  such that  $\xi_i \in \mu(\eta_i)$  and  $\mu''^{-1}(\xi_i) = \mu^{-1}(\eta_i) = \zeta_i$ . Then by virtue of definition 3.2  $q = x_0 \eta_0 x_1 \dots \eta_k x_{k+1} \rightarrow x_{k+2} \eta_{k+1} \dots \eta_n x_{n+2}$  is a production of  $F'$  and  $p \in \mu(q)$ . This implies that  $P'' \subseteq \tilde{P}$ . That is,  $\tilde{F} = F''$ .

Hence the result.

The following theorem illustrates that inflation operators have a maximum property in a sense.

### THEOREM 3.3

For each grammar form  $F$  and for each positive integer  $k$

$$G_1(INF_k(F)) = G_k(F).$$

#### Proof

By virtue of definitions 3.1, 3.2 and by remark 3.1 we obtain that for each grammar form  $F$   $G_1(INF_k(F)) \subseteq G_k(F)$  holds. We show that the reverse inclusion is true, too. Let  $j$  be a positive integer, where  $j \leq k$ . Let  $F' = (V', S', P', \sigma')$  given by a  $j$ -bounded interpretation  $I' = (\mu, V', S', P', \sigma')$  be a  $j$ -bounded interpretation grammar of an arbitrary grammar form  $F = (V, S, P, \sigma)$ . We shall construct a  $k$ -th inflation grammar  $F'' = (V'', S, P'', \sigma'')$  of  $F$  such that  $F'$  is a 1-bounded interpretation grammar of  $F''$ . Let us define a substitution  $\mu'$  on  $V^*$  as follows:

- /i/ for each terminal  $a$  of  $F$  let  $\mu'(a) = \{a\}$ ;
- /ii/ for each inner nonterminal  $\xi$  of  $F$  let  $\mu'(\xi)$  be the union of the following two sets:
  - /1/ a subset of  $V - \Sigma$  consisting of  $k$  elements, such that it contains each element of  $\mu(\xi)$  which is



an inner nonterminal of  $F'$  and it does not contain any other element of  $\mu(\xi)$  and, second,

/2/ the maximal subset of  $\mu(\xi)$  consisting of not inner nonterminals of  $F'$ ;

/iii/ for  $\sigma$  let  $\mu'(\sigma)$  be the union of the following two sets:

/1/ a subset of  $V-\Sigma$  consisting of  $k$  elements such that it contains each element of  $\mu(\sigma)$  which is an inner nonterminal of  $F'$ , and if  $\sigma$  is an inner symbol of  $F$ , then it contains  $\sigma'$  and it does not contain other element of  $\mu(\sigma)$  and, second,

/2/ the maximal subset of  $\mu(\sigma)$  consisting of not inner nonterminals of  $F'$ ;

/iv/ if  $F'$  is not a symbol-tight grammar form then for each outcast nonterminal  $\xi \neq \sigma$  of  $F$  let  $\mu'(\xi) = \mu(\xi)$ .

Let us denote  $V''-S = \bigcup_{\xi \in V-S} \mu'(\xi)$ . Let  $P''$  be the maximal subset of  $\mu'(P)$  such that each element of  $V-\Sigma$  which occurs in at least one element of  $P''$  is an inner nonterminal of  $F'$  or is  $\sigma'$  or is not a symbol of  $F'$ . Clearly,  $I'' = (\mu, V'', S, P'', \sigma')$  is an interpretation of  $F'$ . By virtue of the definition of  $\mu'$  and that of remark 3.1 we obtain that  $I''$  satisfies the conditions of definition 3.2.. Thus  $F'' = (V'', S, P'', \sigma')$  is a  $k$ -th inflation grammar of  $F$ . We have to prove that  $F'$  is a 1-bounded interpretation grammar of  $F''$ . Let us define a substitution  $\mu''$  on  $V''^*$  such as follows:

/i/ for each terminal  $a$  of  $F''$  let  $\mu''(a) = \mu(a)$ ;

/ii/ for each nonterminal  $\xi$  of  $F''$  let  $\mu''(\xi) = \{\xi\}$ .

It is clear, that  $V'-S' \subseteq \bigcup_{\xi \in V''-S} \mu''(\xi)$ . We have to prove that

$P' \subseteq \mu''(P'')$ . Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V'-S'$  / $0 \leq i \leq n$ /,  $x_j \in S'^*$  / $0 \leq j \leq n+2$ /,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary element of  $P'$ . Then there exists a production

$q = y_0 \eta_0 y_1 \dots \eta_k y_{k+1} \rightarrow y_{k+2} \eta_{k+1} \dots \eta_n y_{n+2}$  in  $P$ , where  $\eta_i \in V-S$



$/0 \leq i \leq n/, y_j \in S^* /0 \leq j \leq n+2/, such that  $p \in \mu(q)$  holds. By the definition of  $\mu'$  we obtain that$

$\{y_0 \mu'(\eta_0) y_1 \dots \mu'(\eta_k) y_{k+1} \rightarrow y_{k+2} \mu'(\eta_{k+1}) \dots \mu'(\eta_n) y_{n+2}\} \subseteq P''$ ,  
and by the definition of  $\mu''$

$p \in \{\mu''(y_0) \mu'(\eta_0) \mu''(y_1) \dots \mu'(\eta_k) \mu''(y_{k+1}) \rightarrow \mu''(y_{k+1}) \mu'(\eta_{k+1}) \dots$   
 $\dots \mu'(\eta_n) \mu''(y_{n+2})\} \subseteq \mu''(P'')$ .

This implies that  $P' \subseteq \mu''(P'')$ . Since the other conditions of definition 2.2 hold, therefore  $\tilde{I} = (\mu'', V; S; P; \sigma)$  is an interpretation of  $F''$ . Moreover, it comes with simple considerations that  $\tilde{I}$  satisfies the conditions of definition 3.1 for  $k=1$ . Thus  $F'$  is 1-bounded interpretation grammar of  $F''$ . Since  $F'$  was arbitrary, we proved that for each grammar form  $F$   $G_k(F) \subseteq G_1(INF_k(F))$ . Consequently,  $G_1(INF_k(F)) = G_k(F)$ . Hence the result.

The next theorem comes from theorems 3.1, 3.3 and definition 3.1 with simple applications.

#### THEOREM 3.4

For each grammar form  $F$  and for each positive integer  $k$  and  $l$

$$G_{kl}(F) = G_k(G_l(F))$$

Before the next theorem we need an extension of the notion of the  $k$ -bounded interpretation operator.

The mapping  $G_0$  from  $\mathcal{F}$  into  $2^{\mathcal{F}}$  is called a 0-bounded interpretation operator and is defined as follows: for each grammar form  $F$   $G_0(F) = \emptyset$ . We extend this definition to the subsets of  $\mathcal{F}$  in the following way: for each subset  $\mathcal{F}_1$  of let  $G_0(\mathcal{F}_1) = \emptyset$ .

#### LEMMA 3.1

The elements of  $\tilde{G} = \{G_k | G_k \text{ is a } k\text{-bounded interpretation operator, } k \text{ is a nonnegative integer}\}$  form, with respect to operation  $\cup$  defined by [1] on  $2^{\mathcal{F}}$  a commutative monoid.



/1/ For each subset  $\mathcal{F}_1$  of  $\mathcal{F}$  and for all positive integers  $k$  and  $l$   $(G_k \cup G_l)(\mathcal{F}_1) = G_{\max\{k, l\}}(\mathcal{F}_1)$

The proof is evident, therefore is omitted.

We obtained a theorem expressing an important algebraic property of the bounded interpretation operators.

### THEOREM 3.5

The elements of  $\tilde{G} = \{G_k \mid G_k \text{ is a } k\text{-bounded interpretation operator, } k \text{ is a nonnegative integer}\}$  form, with respect to composition and with respect to operation  $\cup$ , on  $2^{\mathcal{F}}$  a commutative semiring.

### Proof

By lemma 3.1  $\tilde{G}$  is with respect to operation  $\cup$  a commutative monoid with unit element  $G_0$ . By theorem 3.4 and by the fact that  $G_k(G_0(\mathcal{F}_1)) = G_0(G_k(\mathcal{F}_1)) = \emptyset$  for each subset  $\mathcal{F}_1$  in  $\mathcal{F}$  and for each positive integer  $k$  we obtain that  $\tilde{G}$  is with respect to composition a commutative monoid with unit element  $G_1$ .

We have to prove that the distribution laws hold. For each  $G_k, G_l$  and  $G_m$  in  $\tilde{G}$ , where  $k, l, m$  are arbitrary nonnegative integers and for each subset  $\mathcal{F}_1$  in  $\mathcal{F}$  hold that

$$\begin{aligned} (G_m(G_k \cup G_l))(\mathcal{F}_1) &= G_{m \cdot \max\{k, l\}}(\mathcal{F}_1) = G_{\max\{k, l\} \cdot m}(\mathcal{F}_1) = \\ G_{\max\{km, lm\}}(\mathcal{F}_1) &= ((G_k \circ G_m) \cup (G_l \circ G_m))(\mathcal{F}_1). \end{aligned}$$

Since for each subset  $\mathcal{F}_1$  in  $\mathcal{F}$  and for each  $G_k$  in  $\tilde{G}$ , where  $k$  is an arbitrary nonnegative integer  $G_0(G_k(\mathcal{F}_1)) = G_0(\mathcal{F}_1) = \emptyset$  and  $G_k(G_0(\mathcal{F}_1)) = G_0(\mathcal{F}_1) = \emptyset$ , therefore the result is proved.

Next we state a lemma concerning the commutativity of the  $k$ -bounded interpretation operator with the interpretation operator and the  $k$ -th inflation operator and that of the  $k$ -th inflation operator with the interpretation operator.



LEMMA 3.2

For each grammar form  $F$  and for each positive integer  $k$

- /i/  $G(G_k(F)) = G_k(G(F)) = G(F)$ ;
- /ii/  $INF_k(G_l(F)) \subseteq G_l(INF_k(F))$ ; where  $l$  is an arbitrary positive integer;
- /iii/  $INF_k(G(F)) \subseteq G(INF_k(F))$ .

The proofs are simple consequences of the preceeding definitions and theorems, therefore they are omitted.

4.

COMMUTATIVITY OF  $PSE$ ,  $SKEL$ ,  $G_k$   
AND  $INF_k$

We first present the notion of the pseudo-isomorphism operator and that of the skeleton operator. We state some simple relationships concerning these operators. After this we characterize the connection of the pseudo-isomorphism operator with the  $k$ -bounded interpretation operator and the  $k$ -th inflation operator. Finally, we present statements concerning the connection of the skeleton operator with the  $k$ -bounded interpretation operator and the  $k$ -th inflation operator.

DEFINITION 4.1

A grammar form  $F_1 = (V_1, S_1, P_1, \sigma_1)$  is said to be pseudo-isomorphic to a grammar form  $F_2 = (V_2, S_2, P_2, \sigma_2)$  if there exists a pair of mappings  $(v, \tilde{v})$  such that the following conditions hold:

- /i/  $v$  is a one-to-one mapping from  $V_1 - S_1$  onto  $V_2 - S_2$  such that  $v(\sigma_1) = \sigma_2$ ;



/iii/  $\tilde{v}$  is a one-to-one mapping from  $P_1$  onto  $P_2$  such that for all production  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  in  $P_1$ , where  $\xi_i \in V_1 - S_1$  /  $0 \leq i \leq n$  /,  $x_j \in S^*$  /  $0 \leq j \leq n+2$  /,  $n \geq 0$ ,  $0 \leq k \leq n$ , the following holds :

$\tilde{v}(p) = y_0 \eta_0 y_1 \dots \eta_k y_{k+1} \rightarrow y_{k+2} \eta_{k+1} \dots \eta_n y_{n+2}$ , where  $\eta_i \in V_2 - S_2$  /  $0 \leq i \leq n$  /,  $y_j \in S_2^*$  /  $0 \leq j \leq n+2$  /, is a production in  $P_2$  such that  $\eta_i = v(\xi_i)$  /  $0 \leq i \leq n$  / and  $y_j = \varepsilon$  if and only if  $x_j = \varepsilon$  for each  $j$ , where  $0 \leq j \leq n+2$ .

#### DEFINITION 4.2

Let  $F = (V, S, P, \sigma)$  be a grammar form. A binary relation  $\geq_t$  on  $P$  is defined as follows:

a production  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V - S$  /  $0 \leq i \leq n$  /,  $x_j \in S^*$  /  $0 \leq j \leq n+2$  /,  $n \geq 0$ ,  $0 \leq k \leq n$ , is said to be  $t$ -stronger than a production  $q = y_0 \xi_0 y_1 \dots \xi_k y_{k+1} \rightarrow y_{k+2} \xi_{k+1} \dots \xi_n y_{n+2}$ , where  $y_j \in S^*$  /  $0 \leq j \leq n+2$  /, - symbollically  $p \geq_t q$  -, if  $\text{lenght}(x_j) \geq \text{lenght}(y_j)$  for each  $j$ , where  $0 \leq j \leq n+2$ . A production  $p$  of a grammar form  $F$  is said to be maximal with respect to the relation  $\geq_t$  on  $P$  if there is not a production in  $P$  which is  $t$ -stronger than  $p$  or if for each production  $r$  in  $P$  which is  $t$ -stronger than  $p$  and it is different from  $p$  holds that  $p$  is  $t$ -stronger than  $r$ .

The next concept is a simple modification of the concept of a grammar form completed with respect to terminals.

#### DEFINITION 4.3

Let  $a$  be a fixed element of  $\Sigma$ . Let  $I = (\mu, V', \{a\}, P'', \sigma')$  be an interpretation of the grammar form  $F = (V, S, P, \sigma)$  such that the following conditions hold:

- /i/  $\mu(b) = \{a, \varepsilon\}$  for each terminal  $b$  of  $F$ ;
- /ii/  $\mu(\xi) = \{\xi\}$  for each nonterminal  $\xi$  of  $F$ ;
- /iii/  $V' - \{a\} = \bigcup_{\xi \in V - S} \mu(\xi)$ ;



/iv/  $P''$  consists of those elements of  $\mu(P)$  which have the form  $x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V' - \{a\} / 0 \leq i \leq n /$ ,  $x_j \in \{a, \varepsilon\} / 0 \leq j \leq n+2 /$ ,  $n \geq 0$ ,  $0 \leq k \leq n$ .

The grammar form  $F' = (V', \{a\}, P', \sigma)$ , where  $P'$  is the set of those productions of  $P''$  which are maximal with respect to the relation  $\geq_t$  on  $P''$ , is said to be the skeleton grammar of  $F$  and the interpretation  $I' = (\mu, V', \{a\}, P', \sigma')$  is said to be the skeleton interpretation of it.

#### DEFINITION

A grammar form is said to be a skeleton grammar if it is the skeleton grammar of itself.

Next we present the corresponding operators.

#### DEFINITION 4.4

The mapping  $PSE$  and  $SKEL$  from  $\mathcal{F}$  into  $2^{\mathcal{F}}$  are said to be the pseudo-isomorphism operator and the skeleton operator, respectively, if for each grammar form  $F$

$PSE(F) = \{F' \mid F' \text{ is pseudo-isomorphic to } F\}$ ;

$SKEL(F) = \{F' \mid F' \text{ is the skeleton grammar of } F\}$ .

We extend the definition of these mappings to the subsets of  $\mathcal{F}$  in the following way: for each nonvoid subset  $\mathcal{F}_1$  of  $\mathcal{F}$  let  $PSE(\mathcal{F}_1) = \bigcup_{F \in \mathcal{F}_1} PSE(F)$ ,  $SKEL(\mathcal{F}_1) = \bigcup_{F \in \mathcal{F}_1} SKEL(F)$  and let  $PSE(\emptyset) = \emptyset$ ,

$SKEL(\emptyset) = \emptyset$ .

These operators are idempotent mappings, too.

#### THEOREM 4.1

For each grammar form  $F$

/i/  $PSE^2(F) = PSE(F)$ ;

/ii/  $SKEL^2(F) = SKEL(F)$ .

The proofs are direct consequences of the corresponding definitions, therefore are omitted.



We continue by giving some simple relationships. Since the proofs need only simple considerations and some technical details, therefore are omitted.

#### THEOREM 4.2

For each grammar form  $F$

- /i/  $SKEL(PSE(F)) \subseteq PSE(SKEL(F))$ ;
- /ii/  $SKEL(G(F)) \subseteq G(SKEL(F))$ ;
- /iii/  $G(PSE(F)) = PSE(G(F))$ .

Next we characterize the connection of the pseudo-isomorphism operator with the  $k$ -bounded interpretation operator and the  $k$ -th inflation operator.

#### THEOREM 4.3

For each grammar form  $F$  and for each positive integer  $k$

- /i/  $INF_k(PSE(F)) \subseteq PSE(INF_k(F))$  and
- /ii/  $G_k(PSE(F)) = PSE(G_k(F)) = G_k(F)$ .

#### Proof of /i/

Let  $F = (V, S, P, \sigma)$  and  $F_1 = (V_1, S_1, P_1, \sigma_1)$  be pseudo-isomorphic grammar forms. Let  $(\nu, \tilde{\nu})$  be a pair of mappings defining pseudo-isomorphism between  $F$  and  $F_1$ . Suppose that  $F'_1 = (V'_1, S_1, P'_1, \sigma'_1)$  given by a  $k$ -th inflation  $I_1 = (\mu_1, V'_1, S_1, P'_1, \sigma'_1)$  is an arbitrary  $k$ -th inflation grammar of  $F_1$ . We show that there exists a  $k$ -th inflation grammar of  $F$  which is pseudo-isomorphic to  $F'_1$ . Let us define a substitution  $\mu$  on  $V^*$  such as follows:

- /i/ for each terminal  $a$  of  $F$  let  $\mu(a) = \{a\}$ ;
- /ii/ for each nonterminal  $\xi$  of  $F$  let  $\mu(\xi) = \mu_1(\nu(\xi))$ .

Let  $V' = S \bigcup_{\xi \in V-S} \mu(\xi)$ . Let  $P'$  be the maximal subset of  $\mu(P)$  such that each element of  $V-S$  which occurs in at least one element of  $P'$  is an inner nonterminal of  $F'_1$ . It is evident that grammar form  $F' = (V', S, P', \sigma'_1)$  is an interpretation grammar of



$F$  and  $I=(\mu, V'_1, S, P'_1, \sigma'_1)$  is an interpretation of that. Otherwise, it seems that  $\mu$  and  $F'$  satisfy the conditions of definition 3.2. Thus  $F'$  is a  $k$ -th inflation grammar of  $F$  and  $I=(\mu, V'_1, S, P'_1, \sigma'_1)$  is a  $k$ -th inflation of it. In order to complete the proof we have to show that  $F'_1$  and  $F'$  are pseudo-isomorphic grammar forms. We present a pair of mappings  $(\nu, \tilde{\nu})$  between  $F'_1$  and  $F'$  which satisfies the conditions of definition 2.3.. Let  $\nu$  be a mapping from  $V'_1 - S_1$  onto  $V' - S$  such that for each nonterminal  $\xi$  of  $F'_1$   $\nu(\xi) = \{\xi\}$ . It is obvious, that  $\nu$  satisfies the condition /i/ of definition 2.3.. Since  $P'_1$  and  $P$  have the same cardinality, therefore by the definition of  $\nu, \mu$  and  $\tilde{\nu}$  we get that  $P'_1$  and  $P'$  have the same cardinality, too. Let  $\tilde{\nu}$  be a mapping from  $P'_1$  onto  $P'$  such as follows:

for each production  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$  in  $P'_1$ , where  $\xi_i \in V'_1 - S_1$  /  $0 \leq i \leq n$  /,  $x_j \in S_1^*$  /  $0 \leq j \leq n+2$  /,  $n \geq 0$ ,  $0 \leq k \leq n$ , let  $\tilde{\nu}(p) = q = y_0 \xi_0 y_1 \dots \xi_k y_{k+1} \rightarrow y_{k+2} \xi_{k+1} \dots \xi_n y_{n+2}$  be an element of  $P'$ , where  $y_j \in S^*$  /  $0 \leq j \leq n+2$  /, such that  $\mu^{-1}(q) = \tilde{\nu}(\mu_1^{-1}(p))$  holds. It is easy to verify that  $\tilde{\nu}$  is a one-to-one mapping. Since  $\tilde{\nu}$  satisfies the condition /ii/ of definition 2.3, therefore  $\tilde{\nu}$  satisfies it, too. Hence, the pair of mappings  $(\nu, \tilde{\nu})$  defines pseudo-isomorphism between  $F'_1$  and  $F'$ . This means that for each grammar form  $F$

$$INF_k(PSE(F)) \subseteq PSE(INF_k(F)).$$

Hence the result.

The proof of /ii/ comes from theorems 3.1, 3.4 and definitions 2.3, 3.1 with simple considerations.

Next we establish two statements concerning the relation of the operator of completion with respect to terminals to the  $k$ -bounded interpretation operator and the  $k$ -th inflation operator.



#### THEOREM 4.4

For each grammar form  $F$  and for each positive integer  $k$

- /i/  $SKEL(INF_k(F)) = INF_k(SKEL(F))$ ;
- /ii/  $SKEL(G_k(F)) \subseteq G_k(SKEL(F))$ .

#### Proof of /i/

Let  $F_1 = (V_1, S, P_1, \sigma_1)$  given by a  $k$ -th inflation  
 $I_1 = (\mu_1, V_1, S, P_1, \sigma_1)$  be an arbitrary  $k$ -th inflation grammar  
of a grammar form  $F = (V, S, P, \sigma)$ . Let  $F'_1 = (V'_1, \{a\}, P'_1, \sigma_1)$   
and  $F' = (V', \{a\}, P', \sigma)$  be the skeleton grammars of  $F_1$  and  $F$ ,  
respectively. Let  $I'_1 = (\mu'_1, V'_1, \{a\}, P'_1, \sigma_1)$  and  
 $I' = (\mu', V', \{a\}, P', \sigma)$  be the skeleton interpretations of  $F_1$  and  
 $F$ , respectively. We show that  $F'_1$  is a  $k$ -th inflation  
grammar of  $F'$ . Let us define a substitution  $\bar{\mu}'$  on  $V'^*$  as  
follows:

- /i/ let  $\bar{\mu}'(a) = \{a\}$ ; and, /ii/ let  $\bar{\mu}'(\xi) = \mu_1(\xi)$  for each  
nonterminal  $\xi$  of  $F'$ . It is clear that  $V'_1 - \{a\} \subseteq \bigcup_{\xi \in V'^2 - \{a\}} \bar{\mu}'(\xi)$ .

Let  $\bar{P}'$  be the maximal subset of  $\bar{\mu}'(P)$  such that each element  
of  $V - \Sigma$  which occurs in at least one element of  $\bar{P}'$  is an  
inner nonterminal of  $F'_1$ . It comes with simple considerations  
that  $\bar{I}' = (\bar{\mu}', V'_1, \{a\}, \bar{P}', \sigma_1)$  is an interpretation of  $F'$ ; therefore  
 $\bar{F}' = (V'_1, \{a\}, \bar{P}', \sigma_1)$  is an interpretation grammar of it. By  
virtue of remark 3.1 and that of the definitions of  $\mu_1$  and  $\bar{\mu}'$   
we obtain that  $\bar{F}'$  is a  $k$ -th inflation grammar of  $F'$  and  $\bar{I}'$  is  
a  $k$ -th inflation of it. We show that  $\bar{F}' = F'_1$ . It is enough  
to prove that  $\bar{P}' = P_1$ . Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots$   
 $\dots \xi_n x_{n+2}$ , where  $\xi_i \in V'_1 - \{a\}$  /  $0 \leq i \leq n$  / ,  $x_j \in \{a, \epsilon\}$  /  $0 \leq j \leq n+2$  / ,  
 $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary production in  $P_1$ . Then there  
exists a production  $q = y_0 \eta_0 y_1 \dots \eta_k y_{k+1} \rightarrow y_{k+2} \eta_{k+1} \dots \eta_n y_{n+2}$   
in  $P$ , where  $\eta_i \in V - S^*$  /  $0 \leq i \leq n$  / ,  $y_j \in S^*$  /  $0 \leq j \leq n+2$  / such that  
 $p \in \mu'_1(\mu_1(q))$ . By virtue of the definitions of  $\mu'_1$  and  $\mu_1$   
 $p \in \mu'_1(y_0) \mu_1(\eta_0) \mu'_1(y_1) \dots \mu_1(\eta_k) \mu'_1(y_{k+1}) \rightarrow \mu'_1(y_{k+2}) \mu_1(\eta_{k+1}) \dots$   
 $\mu_1(\eta_n) \mu'_1(y_{n+2})$ . Since by the definitions of  $\mu'$  and  $\bar{\mu}'$  and that  
of  $\mu'_1$  and  $\mu_1$   $p \in \mu'(y_0) \bar{\mu}'(\eta_0) \mu'(y_1) \dots \bar{\mu}'(\eta_k) \mu'(y_{k+1}) \rightarrow \mu'(y_{k+2}) \bar{\mu}'(\eta_{k+1}) \dots$   
 $\dots \bar{\mu}'(\eta_n) \mu'(y_{n+2})$ , that is,  $p \in \bar{\mu}'(\mu'(q))$ .



By the definition of  $\bar{P}'$  this implies that  $p \in \bar{P}'$ . Since  $p$  was an arbitrary element of  $P'_1$ , therefore  $P'_1 \subseteq \bar{P}'$ . Let us assume the reverse inclusion. Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V'_1 - \{a\}$  /  $0 \leq i \leq n$  /,  $x_j \in \{a, \epsilon\}$  /  $0 \leq j \leq n+2$  /,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary element of  $\bar{P}'$ . Then there exists a production  $q = y_0 \eta_0 y_1 \dots \eta_k y_{k+1} \rightarrow y_{k+2} \eta_{k+1} \dots \eta_n y_{n+2}$  in  $P$ , where  $\eta_i \in V - S$  /  $0 \leq i \leq n$  /,  $y_j \in S^*$  /  $0 \leq j \leq n+2$  / such that  $p \in \bar{\mu}'(\mu'(q))$ . By virtue of the definition of  $\bar{\mu}'$  and that of  $\mu'$  this means that  $p \in \mu'(y_0) \bar{\mu}'(\eta_0) \mu'(y_1) \dots \bar{\mu}'(\eta_k) \mu'(y_{k+1}) \rightarrow \mu'(y_{k+2}) \bar{\mu}'(\eta_{k+1}) \dots \dots \bar{\mu}'(\eta_n) \mu'_1(y_{n+2})$ . By the definitions of  $\mu'_1$  and  $\mu_1$  and that of  $\bar{\mu}'$  and  $\mu'$  it follows that  $p \in \mu'_1(y_0) \mu_1(\eta_0) \mu'_1(y_1) \dots \dots \mu'_1(\eta_k) \mu_1(y_{k+1}) \rightarrow \mu_1(y_{k+2}) \mu_1(\eta_{k+1}) \dots \mu'_1(\eta_n) \mu_1(y_{n+2})$ . Thus  $p \in \mu'_1(\mu_1(q))$ . By the definition of  $P'_1$  this implies that  $p \in P'_1$ . Since  $p$  was an arbitrary element of  $\bar{P}'$ , therefore  $P'_1 \supseteq \bar{P}'$ . Thus  $P'_1 = \bar{P}'$ . That is,  $F'_1 = \bar{F}'$ . So we proved that  $SKEL(INF_k(F)) \subseteq \subseteq INF_k(SKEL(F))$ . In order to complete the proof we have to show that the reverse inclusion holds. Let  $F' = (V', \{a\}, P', \sigma)$  given by the skeleton interpretation  $I' = (\mu', V', \{a\}, P', \sigma)$  be the skeleton grammar of an arbitrary grammar form  $F = (V, S, P, \sigma)$ . Let  $\bar{F}' = (\bar{V}', \{a\}, \bar{P}', \sigma)$  given by a  $k$ -th inflation  $\bar{I}' = (\bar{\mu}', \bar{V}', \{a\}, \bar{P}', \sigma)$  be a  $k$ -th inflation grammar of  $F'$ . We show that there exists a  $k$ -th inflation grammar  $F_1 = (V_1, S, P_1, \sigma_1)$  of  $F$  such that  $F'_1$  is the skeleton grammar of it. Let us define a substitution  $\mu_1$  on  $V^*$  such as follows:

- /i/ let  $\mu_1(a) = \{a\}$  for each terminal  $a$  of  $F$  and
- /ii/ let  $\mu_1(\xi) = \mu'_1(\xi)$  for each nonterminal  $\xi$  of  $F$ .

It is clear that  $\bar{V}' - \{a\} \subseteq \bigcup_{\xi \in V - S} \mu_1(\xi)$ . Let us denote

$V_1 = S \cup (V'_1 - \{a\})$ . Let  $P_1$  be the maximal subset of  $\mu_1(P)$  such that each element of  $V - \Sigma$  which occurs in at least one element of  $P_1$  is an inner nonterminal of  $\bar{F}'$ . It comes immediately that  $F_1 = (V_1, S, P_1, \sigma_1)$  is an interpretation grammar of  $F$  and  $I_1 = (\mu_1, V_1, S, P_1, \sigma_1)$  is an interpretation of it. Since  $F_1$  and  $I_1$  satisfy the conditions of definition 3.2,



therefore  $F_1$  is a  $k$ -th inflation grammar of  $F$  and  $I_1$  is a  $k$ -th inflation of it. We have to show that  $F'$  is the skeleton grammar of  $F_1$ . Let  $I_1' = (\mu_1', V_1', \{a\}, P_1', \sigma)$  be the skeleton interpretation of  $F_1$ . It is obvious, that  $V_1' = V'$ . We prove that  $P_1 = \bar{P}'$ . Let  $p = x_0 \xi_0 x_1 \dots \xi_k x_{k+1} \rightarrow x_{k+2} \xi_{k+1} \dots \xi_n x_{n+2}$ , where  $\xi_i \in V_1' - \{a\}$   $/0 \leq i \leq n/$   $x_j \in \{a, \emptyset\}$   $/0 \leq j \leq n+2/$ ,  $n \geq 0$ ,  $0 \leq k \leq n$ , be an arbitrary production in  $P_1'$ . Then there exists a production  $q = y_0 \eta_0 y_1 \dots \eta_k y_{k+1} \rightarrow y_{k+2} \eta_{k+1} \dots \eta_n y_{n+2}$ , where  $\eta_i \in V - S$   $/0 \leq i \leq n/$ ,  $y_j \in S^*$   $/0 \leq j \leq n+2/$  in  $P$  such that  $p \in \bar{\mu}'(\bar{\mu}(q))$ . Then by the definition of  $\bar{\mu}$  and that of  $\bar{\mu}'$  we get that  $p \in \{\bar{\mu}(y_0) \bar{\mu}'(\eta_0) \bar{\mu}(y_1) \dots \bar{\mu}'(\eta_k) \bar{\mu}(y_{k+1}) \rightarrow \bar{\mu}(y_{k+2}) \bar{\mu}'(\eta_{k+1}) \dots \dots \bar{\mu}'(\xi_n) \bar{\mu}(y_{n+2})\}$ . Then by the definitions of  $\mu_1$  and  $\mu_1'$  and that of  $\bar{\mu}$  and  $\bar{\mu}'$

$$p \in \{\mu_1'(y_0) \mu_1(\eta_0) \mu_1'(y_1) \dots \mu_1(\eta_k) \mu_1'(y_{k+1}) \rightarrow \mu_1'(y_{k+2}) \mu_1(\eta_{k+1}) \dots \dots \mu_1(\eta_n) \mu_1(y_{n+2})\}.$$

This implies that  $p \in \mu_1'(\mu_1(q))$ . By virtue of the definition of  $P_1$  we obtain that  $p \in P_1'$ . Since  $p$  was arbitrary element of  $P$ , therefore  $P' \subseteq P_1$ . Consequently,  $P' = P_1'$ , that is  $F'$  is the skeleton grammar of  $F_1$ . Hence the result.

The proof of /iii/ is an immediate consequence of the corresponding definitions.

## REFERENCES

- [1] A. CREMERS, S. GINSBURG: Context-free grammar forms, Journal of Computer and System Sciences, 11, 86-117 (1975).
- [2] E. CSUHAJ VARJU: Generatív grammatika formák  $k$ -korlátos interpretációról (On  $k$ -bounded interpretations of generative grammar forms) in Hungarian, Dissertation, 1978.
- [3] S. GINSBURG, H.A. MAURER : On strongly equivalent context-free grammar forms. Computing 16, 281-290 (1976).
- [4] G. RÉVÉSZ: Private communication







## ON THE BASIC CONCEPTS OF SDS (SYSTEM DEVELOPMENT SYSTEM)

### Part II.

*Gábor Dávid*

Computer and Automation Institute  
Hungarian Academy of Sciences  
Budapest, Hungary

## 2. FRAMES

According to the previous introduction, a frame is written in the following form:

```
frame frame-name (formal parameters)
parameter-structure-specification

specification-part:
declarations
frame-specification

implementation-part:
algorithm
control-structure

endframe;
```

The *frame*, *specification*, *implementation* and *endframe* are common basic symbols in the specification languages in SDS. The syntactical form of parameter-specification, declaration, frame-specification depends on the specification language used. The form of the algorithm, written in an implementation language, depends on the language but their control-structure and frame-calling form is common in every implementation language.



To every frame the

- processor of the specification-language
- processor of the implementation-language
- knowledge-base
- the frame belonging the system under design

are associated. They vary in time and in the nature of the development systems. When a frame is under preparation, these associated components form a *system-process*, hence the designer should describe these.

Here we shall discuss these parts. Syntactically this is not an exact definition.

## 2.1. INTERFACE-PART

The *interface-part*:

```
frame frame-name (formal input parameters; formal output
                  parameters)
parameter-structure-specification;
```

declares the symbol "frame-name" as a frame, belonging to the system under design. The name should be unique in this system, otherwise it rewrites the original one. The formal parameters are written as a list, (separated by commas).

The parameter-structure-specification is a list of declarations of the structures in the form

```
structure structure-name <{selector:structure-value}>
reference (structure-name) list of names;
```

Elementary structure-names are *bit*, *bitn* (n arbitrary). The names, declared have are represented by the structures associated. Their data-types (like *integer*, *real*) are declared in the specification-part.



Both of the formal parameters and parameter-specification may be empty. Although the parameter-specification is a part of specification, syntactically and semantically, but it is closer to the identification of frames, i.e. of the interface.

## 2.2. SPECIFICATION-PART

The specification-part consists of two parts:

- declaration
- frame-specification

*Declaration-part* is a list of syntactical entities in the form of  
"declaration-name" declaration-body;

where declaration-name is a symbol, identifying the group of declarations written in declaration-body. This group is placed into the knowledge base, and it can be used by other frames.

If the "declaration-name" part is empty, then the declarations in the declaration-body will not be reuseable. If the declaration-body is empty, SDS searches it by name in the knowledge-base and copies it into this frame. If "declaration-name" is not unique, then this description rewrites the original one in this frame.

The group of declarations is a list of declarations

"type-symbol" is a list of symbols,

where type-symbol is one of the acceptable types in the specification-language. (This is also a requirement to the specification-languages: the new, composed types should be constructed at this point together with the operation defined on them.)



In *Structure Logic Language SL* being a logic-language, the basic set of type-symbols is arbitrary (predicate-symbols) and the user may define arbitrary new types with

- *structure* type-name declaration;
- *reference* (type) object-declaration;
- *type function* function-name declaration;
- *formulae* list of formulae;

where the *function* and *formulae* describe the symbolical and semantical contents of those operations which can be applied on objects, declared by *structure* and *reference* (type).

In *Architecture Language AL* the concepts of *class* in SIMULA'67 are realized by *module*. AL is one of the data-flow languages, hence the description of the components is concentrated on the procedure and on the *input* and *output* data and is realized by the component. Basic data-types are arbitrary (but at the end of system-design it should be specified by appropriate combination of *bit* and *time*).

In the specification-part the user declares only those symbols which are used in the *frame-specification*. Frame-specification states what the frame will do. This is written as a state-transformation,  $I \rightarrow O$ , i.e., from which input  $I$ , which output  $O$  will be produced by the execution of the implementation-part of the frame. In SL this is written in the form of

$$S_i \rightarrow S_o$$

where  $S_i$  and  $S_o$  are logical formulas,  $\rightarrow$  means "implies".

$S_i$  describes the input-state, i.e. the relation between the objects at the beginning of the execution and  $S_o$  describes that after the execution.



In AL the "set of assignments" is used. The assignments are in the form of  $v := e$ , where  $v$  - variable name, declared as input or output,  $e$  - expression. In the declarations, the symbols treated as input and output, respectively, should be declared. Expressions are logical and/or arithmetical expressions, using the function-symbols implemented in AL. Every object declared (for example  $\underline{v}$ ) has a *time* component, which can be specified (as  $v.time$ ) and the value of this component, expressed by an assignment relative to the time of the entry. In the input-output-part, the "set of assignments" may be empty. The declarations and frame-specifications in the specification-languages should describe the frame completely. In SL this means that logically the validity of the statement  $S_i \rightarrow S_o$  can be proved and in the case of AL, the specification-processor can interpret the assignments.

The specification-part should not be empty: at least the declarations should be given.

### 2.3. IMPLEMENTATION

In the implementation-part the problem to be solved by the frame is formulated in an implementation language. It consists of a *set of algorithms* and a *control-structure definition*. The algorithms are treated as generalized transitions of Petri nets, and in the implementation-part the control-structure describes the actual Petri-net, i.e. the sequence of the execution of the algorithm.

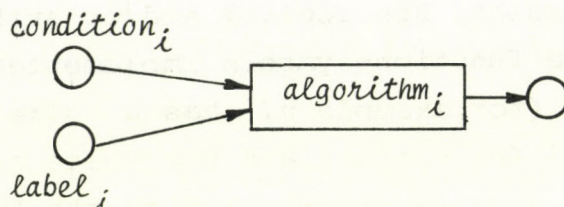
*A l g o r i t h m s.* Each algorithm has a label and a condition.

$$label_i: \quad condition_i \rightarrow algorithm_i$$

where  $label_i$  should be unique within the frame,  $condition_i$  describes the assertion, if it is fulfilled, then the  $algorithm_i$



may be executed. The condition<sub>i</sub> may be the logical constant ( true T). In other words, one can describe an elementary Petri-net in the form:



( controlled later by the control-structure). The elementary Petri-net has

- two head-places  $label_i$  and  $condition_i$ , where  $label_i$  is activated by the control-structure, and the head-place  $condition_i$  generates a token if the condition is true;
- it has a transition, namely  $algorithm_i$  activated if each of its head places has at least one token and whose execution results a token in the tail-place.

The transition  $algorithm_i$  may be a compound one with its control-structure hence structured Petri-nets can be described.

Algorithms are written in the implementation-language. The implementation-languages used allow us to declare and activate two new types: the *frame* and *control*; and symbols, declared as control can be activated on the object, qualified by declaration as *set*. The control-type will be discussed later.

The frames which are declared in the implementation-part, are treated as *local* or *external* ones.

Local frame is described in the implementation-part as a self-contained frame-definition, i.e. just in the form, discussed in this section; it has its own parameters, specification  
- and implementation-part, ended by an *endframe*.



External frames are those symbols, which are declared as

*frame* list of frame-names;

and SDS search for them in the knowledge-base. It should be noted that although the form of the declaration of parameters and external frames is the name, their use and meaning are different: frame, declared as formal parameter is a "variable", having its value (i.e. a frame) at the entry of the frame, but in the implementation-part it is a "literal", identifying the frame.

In the algorithm both local and external frames can be *called*; the form of

frame-name (actual parameters)

where actual parameters should meet the formal ones (in number and type) in the frame-definition. But this is mainly a syntactical requirement, semantically it will be described later in the architecture of SDS, but here the conditions of the call of the frame with specification  $I^* \rightarrow O^*$  in algorithm conditioned by  $cond_i$  can be expressed by the validity of

$$cond_i \Rightarrow I^*$$

where  $\Rightarrow$  represents "implies" (in SL is the same as  $\longrightarrow$ ), and in  $I^*$  the formal parameters are substituted by actual ones.

To explain this, three comments should be made:

- both  $cond_i$  and  $I^*$  may refer to objects which are manipulated by the  $algorithm_i$  just before the call of the frame  $\mathcal{F}^*$ .  $Cond_i$  may state something on these objects and this statement is not necessarily invariant. When the algorithm is initiated, the  $cond_i$  should be true, but executing the  $algorithm_i$ , its logical value



may be changed to false, this does not mean the end of the execution, but if it is false, the algorithm cannot be used as a reentrant program, for example,

- the expression  $cond_i \implies I^*$  should be valid (in the sense of logic), i.e. only those cases are of special interest in which  $cond_i$  is true, because if  $cond_i$  is false, then  $cond_i \implies I^*$  is necessarily true,
- although the  $algorithm_i$  may have its own control-structure, we assume that it is a sequential program. Consequently, the validity of  $cond_i \implies I^*$  can be proved by the analysis of the algorithm before the frame  $I^*$  would be activated.

### Control-structure

The control-structure of a frame can be manipulated object like data objects. In system development in some phases of the design, the control structure is unknown: i.e. how the algorithms will be called in the final version. Control structures can be described in a programming language [6,8] based on the language of Petri-nets designed by Kotov [7]. Here we shall concentrate on the programming language form suitable for frames.

The implementation-part of a frame looks like

$l_1 : cond_1 \rightarrow a_1$

$l_2 : cond_2 \rightarrow a_2$

$\vdots$

$l_n : cond_n \rightarrow a_n$

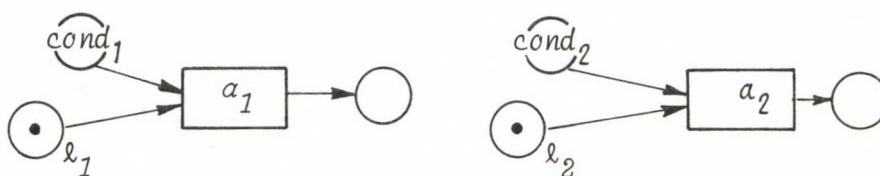
*control-algorithm*

*control-execution*

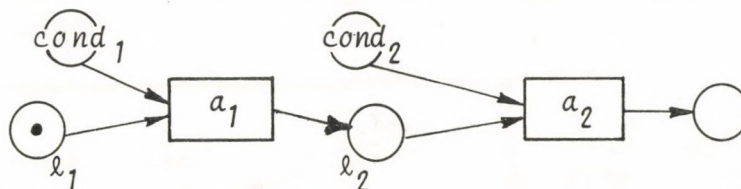


In declaration those symbols should be declared which are local ones. They generally represent data objects. New types: *frame*, *control* and *set*. The *frame* had been previously described. Symbols, declared as *control*, represent control-structure. Control-structure is a well-formed formula defined on the labels  $l_1, l_2, \dots, l$  or on appropriate sets of the labels in the implementation-part with the following operations:

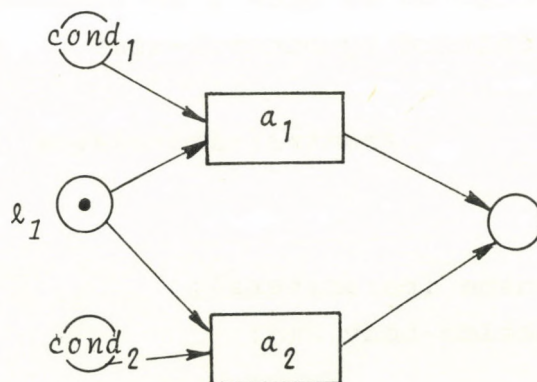
**superposition;** - defines parallel execution, for example  $(l_1, l_2)$



**join;** - defines the sequential execution  $(l_1; l_2)$

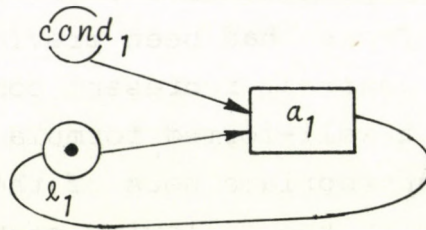


**exclusion:**  $\square$  - defines the exclusive execution  $(l_1 \square l_2)$

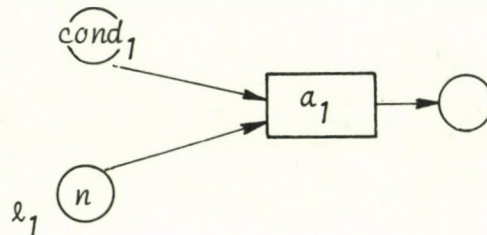




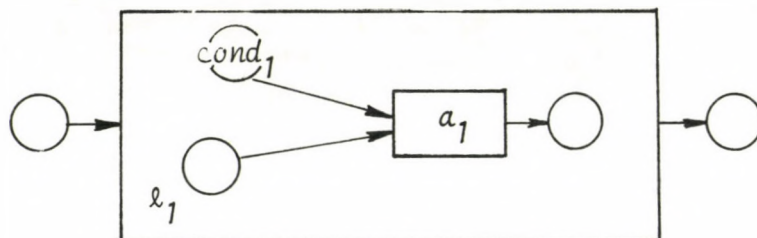
iteration: \* - defines a loop (\*  $l_1$ )



marking:  $n \rightarrow$  places  $n$  tokens into headplace,  $n \rightarrow l_1$



Closure: do ... od closes the Petri-net do  $l_1$  od



Under control-expression we mean a well-formed formula. Control functions can be defined by control-assignments

$c := \text{control-expression}$

and

*control* function-name (parameters);  
begin function-body end;



where the body, consists of an algorithm, composed by control-assignments, control-instructions *for ...; if ... then ... else* and *go to* and arithmetic, logical expressions, written in the specification-language.

Control functions may have sets as parameters. They are specified by the declaration *set*, and the members of the set should be chosen from the labels  $l_1, \dots, l_2, \dots, l_n$ . Set-expression is formed by  $\cup$  (union),  $\cap$  (intersection),  $\sim$  (not) and set-assignment is written as  $s := \text{set-expression}$ , where  $s$  had been declared by a *set*. A label forms a set if bracketed by  $\{ \}$ .

In the implementation-part the control-algorithm computes the sets and the control-structures, but this does not mean the execution of the frame. The control-execution is a well-formed formula defined on labels, on the generated sets and on control-functions. This control-execution will be interpreted by SDS: the system will execute the algorithms in the predescribed order of the control-execution.

In the case of homogeneous algorithms, one may use indices in labels, for example:

$$X[i, j]: \quad i \geq j \rightarrow y \ i := Z[i] * y[j]$$

which should be described only once and defines a set of algorithms, and the indices can be specified in the control-algorithm. Indices are parameters of the algorithms, too.



### 3. ARCHITECTURE OF SDS

In this chapter the realisation of concepts illustrated above will be discussed.

The architecture consists of a set of specification-languages, a set of implementation-languages and system-processors and system-processes written as frames.

Two specification-languages are given:

- SL: Structure Logic Language, designed to describe abstract processors, i.e. abstract data types and transformations defined on them. This is really a *family* of languages, because in SL programming languages can be described (of course, the assembly languages are included, too);
- AL: Architecture Language [6], a language, orthogonal to SL, because it is a *data-flow* language, in which the transformation, performed by architectural components play the most important role and data are used for communication.

The set of implementation-languages covers:

- the assembly-like programming languages
- systems programming languages.

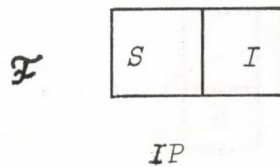
The AL and some of the implementation-languages are in experimental or in design phases.

The relation between specification- and implementation-languages is orthogonal also: the semantics of implementation-languages are given in specification-languages. The Structure Logic Language plays the most important role in this respect; being a logic-based language, it has correct (and adequate) semantics.

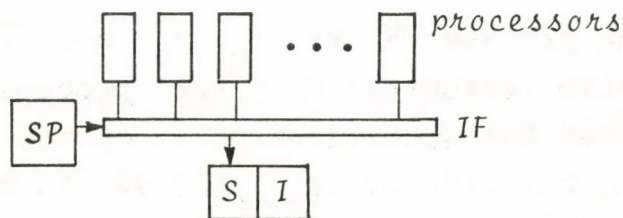
In this section this correspondence will be discussed between the processors/processes and specification/implementation. First we want to describe the static and later the dynamic behaviour of frames.



From the users' point of view, the frame is divided by an interface point IP into two parts: the specification-part  $S$  and implementation-part  $I$



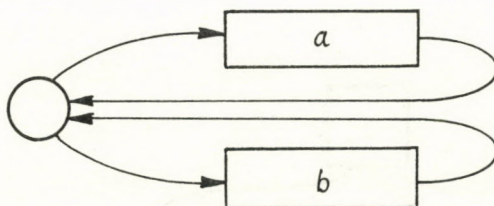
From the other side, system SDS also contains a set of processors, a set of frames and an interface, IF between them. IF is controlled by the processor SP of specification-language.



Allocating a processor to this frame means that the algorithms of implementation-part will be allocated to the processors.

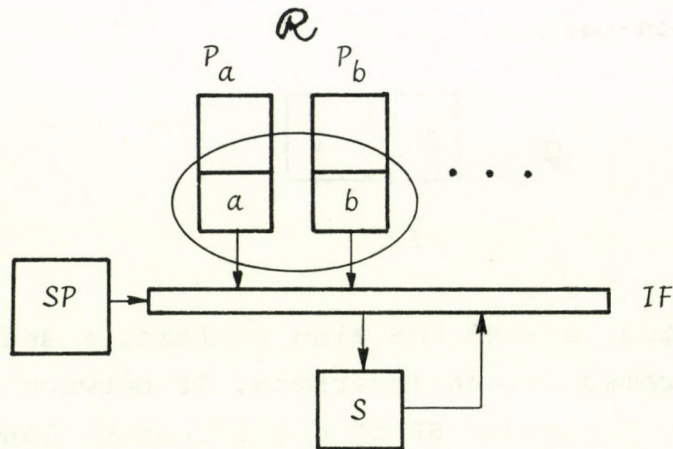
SP interprets the control-algorithm, computes the control-structure and executes that.

The actual "computed" control-structure is kept in the SP and if  $I$  is represented by this control structure  $\mathcal{C}$  and algorithms  $\mathcal{A}$ , for example:



then the system copies the algorithms into the processors (for example  $P_a$  and  $P_b$ ).



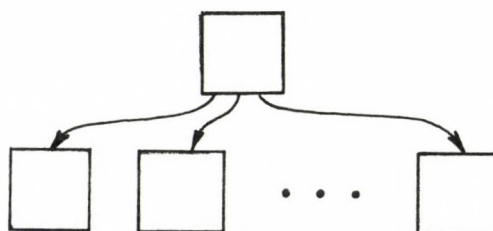


and form the process  $R$  by  $P_a, P_b$  and  $c$ . During the execution of control-structure  $c$ , the process  $R$  is always checked with  $SP$  that the specification  $S$  of the frame  $F$  should be valid during the life of this process. In order to supply a tool for this verification, the user-defined  $IP$  will be copied into the system-interface  $IF$ , where the relevant information can be recognized and compared by  $SP$ .

Dynamically two cases should be analysed:

- during the development of  $F$
- during the development of a system, using  $F$

First, if only the frame  $F$  were designed, its implementation-parts and the processes, formed by its implementation-parts would be different during its development. It means that the frame  $F$  has not only one implementation-part, but a sequence of those:



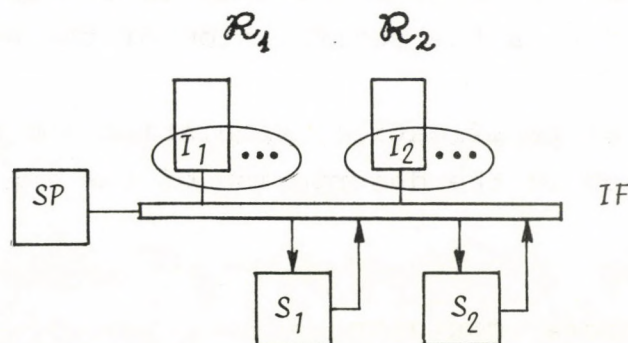


Each of them is intended to fulfill the requirement, described by  $S$ , but they may differ in the implementation tools and languages, they may be

- empty
- different in languages
- realised on different control-structures
- realised hardwarewise

The architecture above makes possible to treat the frame  $\mathcal{F}$  with different implementation-parts, as if it were a unique one, because the specification  $S$  should be invariant against the changes in the implementation-part.

Secondly, during the development of the system, using the frame  $\mathcal{F}$ , we have a structure of frames, calling each other as pre-described in the implementation-parts. If  $\mathcal{F}_1$  calls  $\mathcal{F}_2$ , first the conditions of the activation will be checked by  $SP$ . If the conditions are satisfied, then the implementation-part  $I_2$  of  $\mathcal{F}_2$  will be interpreted (with its control-structure) and allocated on the processor level



The processes  $\mathcal{R}_1$  and  $\mathcal{R}_2$  formed by the appropriate processors and implementation-parts  $I_1$ ,  $I_2$ , together should satisfy the requirements not only of  $S_1$  and  $S_2$ , but the activating-relation between them (described in the previous chapter). Due to the fact that the implementation-part may be different, but the specification is invariant, the change in the implementation-part hasn't any effect on the caller: if



$\mathcal{F}_1$  calls  $\mathcal{F}_2$ , and  $\mathcal{F}_2$  is changed to have implementation-part  $I_2'$ , but  $S_2$  is the same, then the process  $\mathcal{R}_2$  will be the same.

### Processes based on frames

Until now we discussed the frames as they will be executed. But in SDS one may have the following processes formed by frames:

- execution
- change the specification
- change the implementation
- verification
- simulation
- automatic synthesis.

The first one - the execution of a frame - has been described previously. The change of the specification and implementation is not only an editing facility, but in SDS they should be combined with those processes which had been performed on the frame to be changed. This means that if a frame has been already verified, SDS forces the verification of the new version.

The processes of verification, simulation and automatic synthesis are based on the description of the implementation-language.

The *knowledge-base* contains

- the description of the implementation-language
- the specification-parts of frames
- the history of frames, i.e. the former versions of the frame and the processes performed on them.

The description of the implementation-language is given in the specification-language of the frame. This description in the case of Structure Logic SL is a logical one, provided with a



mechanical theorem proving technique, by which properties of the frames can be proved. In the *process of verification*: the implementation- and the specification-part can be compared or the specifications of interconnected frames can be proved (regardless of their implementation-part). In the case of AL, verification is based on topological analysis of the frames used.

*Automatic synthesis* is based on this calculi for theorem proving: if given a frame but with empty implementation-part, then SDS tries to synthetise the algorithm representing the implementation-part. This means that the designer should concentrate on the description of the frames, i.e. on the specifications, and if a hierarchy of frames has been designed, the SDS either may verify or prove the description and may synthetise the necessary algorithm implementing the frames.

The *process of simulation* is based on the description of the implementation-language also, but in Architecture Language AL the implementation-part may be simulated directly. In the case of SL the specification-part supplies the information for simulation.

#### 4. SUMMARY

In this working paper, the concept of a new operating system has been formally introduced. This operating system SDS is specially designed for system-development, where the system to be designed may be a programming and/or computing system (including hardware). SDS is under design, hence further features and languages will be added.



## REFERENCES

- [1] G. Dávid: Structured Automatized Design of Microprograms, in Large Scale Integration, H.W. Lawson et al. (eds.), North-Holland (1978).
- [2] G. Dávid, S. Keresztély and A. Sárközy: Microprogram Synthesis by Theorem Proving. Proceedings of the II. Hungarian Comp. Sci. Conf. (1977), Part, 292-310.
- [3] G. Dávid, S. Keresztély, I. Losonczi and A. Sárközy: Logic-Based Description of Microcomputers (1978) MTA SZTAKI Közlemények, Budapest (Hungary).
- [4] G. Dávid, S. Keresztély, I. Losonczi and A. Sárközy: Microprogram Synthesis (1978) MTA SZTAKI Közlemények, Budapest (Hungary )
- [5] G. Dávid: Proving Correctness and Automatic Synthesis of Parallel programs. 1979. VEDA, Bratislava. (In Russian: Nauka, Novosibirsk (in print)).
- [6] G. Dávid: Architecture Language. MTA SZTAKI Közlemények, Budapest, 1979. (in print).
- [7] V.E. Kotov: Concurrent Programming with Control Types. In: Constructing Quality Software, North-Holland, 1978.
- [8] G. Dávid: Description of Dynamic Control Structures, Algorithms'79. Proceedings.
- [9] D. Warren and L. Pereira: PROLOG - The Language and its Implementation Compared with LISP. Proc. of ACM SIGART-SIGPLAN Conf. on "AI and Programming Languages", Rochester, N.Y. August, 1977.
- [10] R. Kowalski: Predicate Logic as Programming Language, Proc. IFIP Congress, 1974, North-Holland.



- [11] R.M. Burstall and J.A. Gougen: Putting Theories together to make Specifications. Proc. of the 5th IJCAI, Cambridge, Mass. 1977.
- [12] K. Nahijama, M. Honda and H. Nakahara: Describing and Verifying Programs with Abstract Data Types. Proc. of "Formal Description of Programming Concepts", E.J. Heuhold (ed.) North-Holland, 1978.
- [13] W.A. Wulf, R.L. London and M. Shaw: Abstraction and Verification in Alphard: Introduction to Language and Methodology. Techn. Rep. Carnegie-Mellon University, Pittsburgh, Penn. 1976.
- [14] G. Dávid, I. Losonczi, S.D. Papp: Language Support for Designing Multilevel Computer Systems. CONPAR'81, Springer Verlag. LNCS 111, pp. 85-100.
- [15] G. Dávid: Problemsolving = Knowledge + Strategy. Int. Conf. on Artificial Intelligence and Information-Control System of Robots. (in print).



1911 H. M. Bassett and J. A. G. ...  
to make suggestions. ... of ...  
... 1911.

... M. ... and ...  
... ...  
... ...  
... (and) ...

... ...  
... ...  
... ...  
... ...  
... ...  
... ...  
... ...

... ...  
... ...



# ON GENERALIZED ITERATIVE ALGEBRAIC THEORIES

Zoltán Ésik

University of Szeged

Szeged, Hungary

## ABSTRACT

A basis of identities of generalized iterative was given in [2]. Here we present a simplified version of this basis and prove two theorems regarding the connection between iterative and generalized iterative theories.

## 1. INTRODUCTION

This paper is a supplement to [2]. Therefore all notions and notations will be used in accordance with [2].

The main result in [2] was a presentation of a basis of identities of generalized iterative theories. Namely, it was shown that identities from (A) to (E) below, together with those defining algebraic theories, are satisfactory to characterize generalized iterative theories.

$$(A) \quad f \langle f^+, 1_p \rangle = f^+ \quad \text{if} \quad f: n \rightarrow n+p,$$

$$(B) \quad \langle f, g \rangle^+ = \langle h^+, (g\rho)^+ \langle h^+, 1_p \rangle \rangle \quad \text{if} \quad f: n \rightarrow n+m+p,$$

$$g: m \rightarrow n+m+p, \quad h = f \langle 1_n + 0_p, (g\rho)^+, 0_n + 1_p \rangle$$

$$\text{and } \rho = \langle 0_m + 1_n, 1_m + 0_n \rangle + 1_p,$$

$$(C) \quad (0_n + f)^+ = f \quad \text{if} \quad f: n \rightarrow p,$$

$$(D) \quad (f + 0_q)^+ = f^+ + 0_q \quad \text{if} \quad f: n \rightarrow n+p,$$

$$(E) \quad \langle \pi_m^1 \rho g(\rho_1 + 1_p); \dots, \pi_m^m \rho g(\rho_m + 1_p) \rangle^+ = \rho(g(\rho + 1_p))^+$$

$$\text{if } g: n \rightarrow m+p, \quad \rho: m \rightarrow n, \quad \rho_i: m \rightarrow m \quad (i \in [m])$$

$$\text{are base such that } \rho \text{ is surjective, } \rho_i \rho = \rho \quad (i \in [m]).$$

In [2] some consequences of the preceding basis were established. Further on, these consequences will be referred



in accordance with [2].

In Section 2 a somewhat simplified basis will appear. Namely, we show that identities from  $(A_1)$  to  $(F_1)$  give an alternative basis of identities of generalized iterative theories.

- $(A_1)$   $f^+ = f \langle f^+, 1_p \rangle$  if  $f: 1 \rightarrow 1+p$ ,
- $(B_1)$   $\langle f, g \rangle^+ = \langle h^+, (g\rho)^+ \langle h^+, 1_p \rangle \rangle$  if  $f: n \rightarrow n+1+p$ ,  
 $g: 1 \rightarrow n+1+p$ ,  $h = f \langle 1_n + 0_p, (g\rho)^+, 0_{n+1_p} \rangle$  and  
 $\rho = \langle 0_1 + 1_n, 1_1 + 0_n \rangle + 1_p$ ,
- $(C_1)$   $(0_1 + f)^+ = f$  if  $f: 1 \rightarrow p$ ,
- $(D_1)$   $(f(1_1 + \rho))^+ = f^+ \rho$  if  $f: 1 \rightarrow 1+p$ ,  $\rho: p \rightarrow q$  is base
- $(E_1)$   $\pi_m^1 \langle \pi_m^1 \rho g(\rho_1 + 1_p), \dots, \pi_m^m \rho g(\rho_m + 1_p) \rangle^+ = \pi_n^1 (g(\rho + 1_p))^+$   
if  $g: n \rightarrow m+p$ ,  $\rho: m \rightarrow n$  is monotone and surjective,  
and  $\rho_i: m \rightarrow m$  are base with  $\rho_i \rho = \rho$  ( $i \in [m]$ ),
- $(F_1)$   $(\rho f(\rho^{-1} + 1_p))^+ = \rho f^+$  if  $f: 2 \rightarrow 2+p$ ,  $\rho = \langle 0_1 + 1_1, 1_1 + 0_1 \rangle$ .

In Section 3 we present two theorems, regarding the connection between iterative and generalized iterative theories. All results were already noticed in [2].

## 2. A SIMPLIFICATION OF THE BASIS

Lemma 1  $(A_1), (B_1) \models (A)$ .

Proof. This lemma was already discovered in paper [1]. Indeed, the first part of the proof of the main result of [1] can be translated with the exception that  $(B_1)$ , rather than the dual of it, is utilizable here.

Lemma 2  $(C_1), (B_1) \models (C)$ .

Proof. Let  $f: n \rightarrow p$  be an arbitrary element in an algebraic theory with iteration satisfying  $(C_1)$  and  $(B_1)$ . We shall show by induction on  $n$  that  $(0_n + f)^+ = f$ . The case  $n=0$  is obvious. Assume now that  $n>0$  and the proof is done for  $n-1$ .



Define  $f_1 = 0_n + (1_{n-1} + 0_1)f$ ,  $g = 0_n + \pi_n^n f$ .

It is obvious that  $\langle f_1, g \rangle = 0_n + f$ . Let us introduce the following notations:

$$\rho = \langle 0_1 + 1_{n-1}, 1_1 + 0_{n-1} \rangle + 1_p,$$

$$h = f_1 \langle 1_{n-1} + 0_p, (g\rho)^+, 0_{n-1} + 1_p \rangle.$$

An easy computation shows that  $g\rho = g$  and  $h = 0_{n-1} + (1_{n-1} + 0_1)f$ . Hence, by the induction hypothesis,  $h^+ = (1_{n-1} + 0_1)f$ . On the other hand, by  $(B_1)$  and  $(C_1)$ , we obtain  $(0_n + f)^+ = \langle f_1, g \rangle^+ = \langle h^+, (g\rho)^+ \langle h^+, 1_p \rangle \rangle = \langle (1_{n-1} + 0_1)f, (0_{n-1} + \pi_n^n f) \langle h^+, 1_p \rangle \rangle = \langle (1_{n-1} + 0_1)f, \pi_n^n f \rangle = f$ .

Lemma 3  $(D_1), (B_1) \vdash (D)$ .

Proof. First observe that  $(D_1)$  contains identity  $(f + 0_q)^+ = f^+ + 0_q (f: 1 \rightarrow 1+p)$  as a special case. Indeed,  $(f + 0_q)^+ = (f(1_1 + 1_p + 0_q))^+ \stackrel{(D_1)}{=} f^+(1_p + 0_q) = f^+ + 0_q$ . With this observation in mind, the proof can be carried out by a simple induction on  $n$ .

In order to simplify the formalization of the proof of the next lemma, let us introduce the following notations. Let  $n_1, \dots, n_k \geq 0$  ( $k \geq 1$ ) be arbitrary integers, and let  $i_1, \dots, i_k$  be a permutation of  $[k]$  and  $i_0 = 0$ . Then we shall use, ambiguously,

$\theta_{n_1, \dots, n_k}^{n_{i_1}, \dots, n_{i_k}}$  to denote the base morphism, corresponding to the

mapping  $\alpha: [\sum_{i=1}^k n_i] \rightarrow [\sum_{i=1}^k n_i]$  defined by  $i\alpha = j$  if and only

if there exist integers  $t, s \in \{0, \dots, k-1\}$  and  $j_0 \geq 1$  such that

$$i_s = t, j_0 \in [n_{t+1}], i = \sum_{\ell=1}^t n_\ell + j_0, \text{ and } j = \sum_{\ell=1}^s n_{i_\ell} + j_0. \text{ Thus,}$$

for example,  $\theta_{m, n, p}^{n, m, p}$  stands for base morphism, corresponding to the mapping indicated below:



$$\begin{array}{ccc} n & + & m & + & p \\ & \searrow & & \swarrow & \downarrow \\ m & + & n & + & p \end{array}$$

Lemma 4  $(B_1), (D_1) \models (B)$ .

Proof. Let  $f: n \rightarrow n+m+p$  and  $g: m \rightarrow n+m+p$  be arbitrary morphisms in an algebraic theory with iteration which satisfies  $(B_1)$  and  $(D_1)$ . We shall prove by induction on  $m$  that

$$\langle f, g \rangle^+ = \langle h^+, (g\rho)^+ \langle h^+, 1_p \rangle \rangle \quad (1)$$

where

$$h = f \langle 1_{n+0_p}, (g\rho)^+, 0_{n+1_p} \rangle, \quad (2)$$

$$\rho = \theta_{m,n,p}^{n,m,p}. \quad (3)$$

The case  $m=0$  is trivial and for  $m=1$  our statement coincides with  $(B_1)$ . Now let  $m>1$ ,  $m_1=m-1$ ,  $m_2=1$ ,  $g_1 = (1_{m_1} + 0_1)g$ ,  $g_2 = \pi_m^m g$ , and define the morphisms  $h$  and  $\rho$  by (2) and (3), respectively. By  $(B_1)$  we have

$$\langle f, g \rangle^+ = \langle a^+, (g_2\alpha)^+ \langle a^+, 1_p \rangle \rangle \quad (4)$$

where

$$a = \langle f, g_1 \rangle \langle 1_{n+m_1+0_p}, (g_2\alpha)^+, 0_{n+m_1+1_p} \rangle, \quad (5)$$

$$\alpha = \theta_{m_2,n,m_1,p}^{n,m_1,m_2,p}. \quad (6)$$

By the induction hypothesis we get

$$a^+ = \langle b^+, (g_1 \langle 1_{n+m_1+0_p}, (g_2\alpha)^+, 0_{n+m_1+1_p} \rangle \beta)^+ \langle b^+, 1_p \rangle \rangle, \quad (7)$$

$$b = f \langle 1_{n+m_1+0_p}, (g_2\alpha)^+, 0_{n+m_1+1_p} \rangle.$$

$$\cdot \langle 1_{n+0_p}, (g_1 \langle 1_{n+m_1+0_p}, (g_2\alpha)^+, 0_{n+m_1+1_p} \rangle \beta)^+, 0_{n+1_p} \rangle, \quad (8)$$



$$\beta = \theta_{m_1, n, p}^{n, m_1, p} \quad (9)$$

Again, by (B<sub>1</sub>)

$$(g\rho)^+ = \langle c^+, (g_2\rho\gamma)^+ \langle c^+, 1_{n+p} \rangle \rangle \quad (10)$$

where

$$c = g_1 \rho \langle 1_{m_1} + 0_{n+p}, (g_2\rho\gamma)^+, 0_{m_1} + 1_{n+p} \rangle, \quad (11)$$

$$\gamma = \theta_{m_2, m_1, n, p}^{m_1, m_2, n, p} \quad (12)$$

It is easy to check that

$$\rho\gamma = \alpha \theta_{m_2, m_1, n, p}^{m_2, n, m_1, p} \quad (13)$$

Now we show that

$$g_1 \langle 1_{n+m_1} + 0_p, (g_2\alpha)^+, 0_{n+m_1} + 1_p \rangle \beta = c. \quad (14)$$

Indeed,

$$\begin{aligned} & g_1 \langle 1_{n+m_1} + 0_p, (g_2\alpha)^+, 0_{n+m_1} + 1_p \rangle \beta = \quad /by (9)/ \\ & = g_1 \langle 1_{n+m_1} + 0_p, (g_2\alpha)^+, 0_{n+m_1} + 1_p \rangle \theta_{m_1, n, p}^{n, m_1, p} = \\ & = g_1 \langle \theta_{m_1, n, p}^{n, m_1, p}, (g_2\alpha)^+ \theta_{m_1, n, p}^{n, m_1, p}, 0_{n+m_1} + 1_p \rangle = \quad /by (D_1)/ \\ & = g_1 \langle \theta_{m_1, n, p}^{n, m_1, p}, (g_2 \alpha \theta_{m_2, m_1, n, p}^{m_2, n, m_1, p})^+, 0_{n+m_1} + 1_p \rangle = \quad /by (13)/ \\ & = g_1 \langle \theta_{m_1, n, p}^{n, m_1, p}, (g_2\rho\gamma)^+, 0_{n+m_1} + 1_p \rangle = \\ & = g_1 \theta_{m_1, m_2, n, p}^{n, m_1, m_2, p} \langle 1_{m_1} + 0_{n+p}, (g_2\rho\gamma)^+, 0_{m_1} + 1_{n+p} \rangle = c. \quad /by (3), \\ & \quad (11)/ \end{aligned}$$



Substituting (14) in (7) and (8), results

$$a^+ = \langle b^+, c^+ \langle b^+, l_p \rangle \rangle, \quad (15)$$

$$b = f \langle l_{n+m_1}^{+0}, (g_2 \alpha)^+, 0_{n+m_1+1} \rangle \langle l_n^{+0}, c^+, 0_{n+1} \rangle. \quad (16)$$

Next, we show that  $h=b$ . (17)

$$h = \quad \text{/by (2)/}$$

$$= f \langle l_n^{+0}, (g\rho)^+, 0_{n+1} \rangle = \quad \text{/by (10)/}$$

$$= f \langle l_n^{+0}, c^+, (g_2 \rho \gamma)^+ \langle c^+, l_{n+p} \rangle, 0_{n+1} \rangle = \quad \text{/by (13), (D_1)/}$$

$$= f \langle l_n^{+0}, c^+, (g_2 \alpha)^+ \theta_{m_1, n, p}^{n, m_1, p} \langle c^+, l_{n+p} \rangle, 0_{n+1} \rangle =$$

$$= f \langle l_n^{+0}, c^+, (g_2 \alpha)^+ \langle l_n^{+0}, c^+, 0_{n+1} \rangle, 0_{n+1} \rangle =$$

$$= f \langle l_{n+m_1}^{+0}, (g_2 \alpha)^+, 0_{n+m_1+1} \rangle \langle l_n^{+0}, c^+, 0_{n+1} \rangle = b.$$

/by (16)/

Observe that in the course of proving (17) we have actually shown that

$$(g\rho)^+ = \langle c^+, (g_2 \alpha)^+ \theta_{m_1, n, p}^{n, m_1, p} \langle c^+, l_{n+p} \rangle \rangle. \quad (18)$$

Thus,

$$\langle f, g \rangle^+ = \quad \text{/by (4)/}$$

$$= \langle a^+, (g_2 \alpha)^+ \langle a^+, l_p \rangle \rangle = \quad \text{/by (15), (17)/}$$

$$= \langle h^+, c^+ \langle h^+, l_p \rangle, (g_2 \alpha)^+ \langle h^+, c^+ \langle h^+, l_p \rangle, l_p \rangle \rangle =$$

$$= \langle h^+, c^+ \langle h^+, l_p \rangle, (g_2 \alpha)^+ \theta_{m_1, n, p}^{n, m_1, p} \langle c^+, l_{n+p} \rangle \langle h^+, l_p \rangle \rangle =$$



$$\begin{aligned}
 &= \langle h^+, \langle c^+, (g_2 \alpha)^+ \theta_{m_1, n, p}^{n, m_1, p} \langle c^+, l_{n+p} \rangle \rangle \langle h^+, l_p \rangle \rangle = \quad / \text{by (18)} / \\
 &= \langle h^+, (g\rho)^+ \langle h^+, l_p \rangle \rangle,
 \end{aligned}$$

i.e. what was to be proved.

Lemma 5  $(F_1), (B_1), (D_1) \models (F)^*$

Proof. Instead of it we shall prove  $(F_1), (B), (D_1) \models (F)$ .

Observe that if  $\rho, \tau: n \rightarrow n$  are bijective base morphisms, moreover, both  $(\rho f(\rho^{-1} + l_p))^+ = \rho f^+$  and  $(\tau f(\tau^{-1} + l_p))^+ = \tau f^+$  are identities in an algebraic theory with iteration, then also

$(\rho \tau f((\rho \tau)^{-1} + l_p))^+ = \rho \tau f^+$  is an identity in this theory. This observation enables us to prove our statement for bijective base morphisms of type  $l_n + \langle 0_1, +l_1, l_1 + 0_1 \rangle + l_m$  where  $n, m \geq 0$  are arbitrary integers.

Consider now an arbitrary algebraic theory with iteration  $T$ , and let  $\rho: n \rightarrow n$  be bijective,  $f \in T(n, n+p)$  where  $n \geq 2$ ,  $p \geq 0$ . We have to prove that  $(\rho f(\rho^{-1} + l_p))^+ = \rho f^+$ .

The case  $n=2$  is trivial. We proceed by induction on  $n$ . Assume that  $n > 2$ . Let us distinguish two cases.

First suppose that  $\rho = 1_m + \alpha$  where  $m = n-2$ ,  $\alpha = \langle 0_1 + l_1, l_1 + 0_1 \rangle$ . Note that  $\alpha^{-1} = \alpha$  and  $\rho^{-1} = \rho$ .

Let  $a = (1_m + 0_2)f$ ,  $b = (0_m + l_2)f$ ,  $\beta = \langle 0_2 + l_m, l_2 + 0_m \rangle + l_p$ . An easy computation produces  $(\rho + l_p)\beta = \beta(\alpha + l_{m+p})$ . Define  $h, k: m \rightarrow m+p$  by  $h = a \langle 1_m + 0_p, (b\beta)^+, 0_m + l_p \rangle$  and  $k = a(\rho + l_p) \langle 1_m + 0_p, (\alpha b(\rho + l_p)\beta)^+, 0_m + l_p \rangle$ .

We show that  $h = k$ . Indeed, by  $(\rho + l_p)\beta = \beta(\alpha + l_{m+p})$  and  $(F_1)$ , we obtain  $(\alpha b(\rho + l_p)\beta)^+ = \alpha(b\beta)^+$ . Thus,  $k = a(1_m + \alpha + l_p)$ :

$\cdot \langle 1_m + 0_p, \alpha(b\beta)^+, 0_m + l_p \rangle = a \langle 1_m + 0_p, (b\beta)^+, 0_m + l_p \rangle = h$  can be derived.

We have seen that  $h = k$ . Therefore, also  $(b\beta)^+ \langle h^+, l_p \rangle = \alpha(\alpha b(\rho + l_p)\beta)^+ \langle k^+, l_p \rangle$ . From this the result follows by applying (B) twice:

\*Regarding identity (F) cf. [2].



$$\begin{aligned}
 (\rho f(\rho+1_p))^+ &= \langle a(\rho+1_p), ab(\rho+1_p) \rangle^+ = \\
 &= \langle k^+, (ab(\rho+1_p)\beta)^+ \langle k^+, 1_p \rangle \rangle = \\
 &= \langle h^+, \alpha(b\beta)^+ \langle h^+, 1_p \rangle \rangle = \\
 &= \rho \langle h^+, (b\beta)^+ \langle h^+, 1_p \rangle \rangle = \rho f^+.
 \end{aligned}$$

Secondly, assume that  $\rho = \alpha + 1_1$ , where  $\alpha = 1_m + \langle 0_1 + 1_1, 1_1 + 0_1 \rangle + 1_\ell$ :  
 $:n-1 \rightarrow n-1$  for some integers  $m, \ell \geq 0$ . We have again  $\alpha = \alpha^{-1}$  and  $\rho = \rho^{-1}$ .

In this case let  $\beta = \langle 0_1 + 1_{n-1}, 1_1 + 0_{n-1} \rangle + 1_p$ ,  $h = a \langle 1_{n-1} + 0_p, (b\beta)^+, 0_{n-1} + 1_p \rangle$ ,  $k = \alpha a(\rho+1_p) \langle 1_{n-1} + 0_p, (b(\rho+1_p)\beta)^+, 0_{n-1} + 1_p \rangle$  where  $a = (1_{n-1} + 0_1)f$ ,  $b = \pi_n^n f$ . We are going to show that  $k = ah(\alpha+1_p)$ .  
 Indeed, by  $(\rho+1_p)\beta = \beta(1_1 + \alpha+1_p)$  and  $(D_1)$  it follows that  $(b(\rho+1_p)\beta)^+ = (b\beta)^+(\alpha+1_p)$ , hence,

$$\begin{aligned}
 k &= \alpha a(\rho+1_p) \langle 1_{n-1} + 0_p, (b(\rho+1_p)\beta)^+, 0_{n-1} + 1_p \rangle = \\
 &= \alpha a \langle \alpha + 0_p, (b\beta)^+(\alpha+1_p), 0_{n-1} + 1_p \rangle = \\
 &= \alpha a \langle 1_{n-1} + 0_p, (b\beta)^+, 0_{n-1} + 1_p \rangle (\alpha+1_p) = ah(\alpha+1_p).
 \end{aligned}$$

Thus,  $k = ah(\alpha+1_p)$  and by the induction hypothesis,  $k^+ = ah^+$ .  
 Applying  $(B_1)$  twice, this yields

$$\begin{aligned}
 (\rho f(\rho+1_p))^+ &= \langle \alpha a(\rho+1_p), b(\rho+1_p) \rangle^+ = \\
 &= \langle k^+, (b(\rho+1_p)\beta)^+ \langle k^+, 1_p \rangle \rangle = \\
 &= \langle ah^+, (b\beta)^+(\alpha+1_p) \langle ah^+, 1_p \rangle \rangle = \\
 &= \langle ah^+, (b\beta)^+ \langle h^+, 1_p \rangle \rangle = \\
 &= (\alpha+1_1) \langle h^+, (b\beta)^+ \langle h^+, 1_p \rangle \rangle = \rho f^+.
 \end{aligned}$$



This ends the proof of Lemma 5.

Lemma 6  $(E_1), (B_1), (D_1), (F_1) \models (E)$ .

Proof. By virtue of the previous lemma, it is enough to show that  $(E_1), (F) \models (E)$ . Therefore, take an arbitrary theory with iteration  $T$  which satisfies  $(E_1)$  and  $(F)$ .

First we shall prove that the restriction of  $(E)$  to monotone surjections is valid in  $T$ . For this purpose consider an optional morphism  $g: n \rightarrow m+p$  ( $n, m \geq 1, p \geq 0$ ) as well as base morphisms  $\rho: m \rightarrow n$ ,  $\rho_1, \dots, \rho_m: m \rightarrow m$  with  $\rho_i \rho = \rho$  ( $i \in [m]$ ), and assume that  $\rho$  is a monotone surjection. Let  $i \in [m]$  be an arbitrary fixed integer. It is obvious that there exist bijective base morphisms  $\alpha: m \rightarrow m$  and  $\beta: n \rightarrow n$  with the following properties:

(i)  $\alpha \rho \beta$  is monotone, (ii)  $l\alpha = i$ .

Define  $\tau = \alpha \rho \beta$ ,  $\tau_j = \alpha \rho_j \alpha^{-1}$  ( $j \in [m]$ ). By a short computation we get  $\tau_j \tau = \tau$  ( $j \in [m]$ ) and  $\beta^{-1} g(\alpha^{-1} + l_p)(\tau + l_p) = \beta^{-1} g(\rho \beta + l_p)$ . Thus,

$$\begin{aligned} \pi_m^1 < \pi_m^1 \tau \beta^{-1} g(\alpha^{-1} + l_p)(\tau + l_p), \dots, \pi_m^m \tau \beta^{-1} g(\alpha^{-1} + l_p)(\tau + l_p) >^+ = \\ = \pi_m^1 \tau (\beta^{-1} g(\rho \beta + l_p))^+ \end{aligned}$$

follows by  $(E_1)$ . However,  $\tau \beta^{-1} g(\alpha^{-1} + l_p)(\tau + l_p) = \alpha \rho g(\rho_j + l_p)(\alpha^{-1} + l_p)$  ( $j \in [m]$ ) and  $\beta^{-1} g(\rho \beta + l_p) = \beta^{-1} g(\rho + l_p)(\beta + l_p)$ . Hence, two applications of  $(F)$  will yield

$$\pi_m^1 \alpha < \pi_m^1 \rho g(\rho_1 + l_p), \dots, \pi_m^m \rho g(\rho_m + l_p) >^+ = \pi_m^1 \tau \beta^{-1} (g(\rho + l_p))^+.$$

By  $l\alpha = i$  and  $\tau \beta^{-1} = \alpha \rho$ , this means

$$\pi_m^i < \pi_m^1 \rho g(\rho_1 + l_p), \dots, \pi_m^m \rho g(\rho_m + l_p) >^+ = \pi_m^i \rho (g(\rho + l_p))^+.$$

As  $i$  was arbitrary, we obtain that  $(E)$  is valid in  $T$  for any monotone surjective  $\rho$ .

Now let  $\rho$  be an arbitrary surjective base morphism and suppose that  $g$  and  $\rho_i$  ( $i \in [m]$ ) are given as previously. There exists a bijective base morphism  $\alpha: m \rightarrow m$  such that  $\tau = \alpha \rho$  is



monotone. Define  $\tau_i$  by  $\tau_i = \alpha \rho_i \alpha^{-1}$  ( $i \in [m]$ ). It is easy to check to  $\tau_i \tau = \tau$  ( $i \in [m]$ ). As  $\tau$  is monotone and  $g(\alpha^{-1} + 1_p)(\tau + 1_p) = g(\rho + 1_p)$  we get

$$\langle \pi_m^1 \tau g(\alpha^{-1} + 1_p)(\tau_1 + 1_p), \dots, \pi_m^m \tau g(\alpha^{-1} + 1_p)(\tau_m + 1_p) \rangle^+ = \tau(g(\rho + 1_p))^+.$$

But  $\tau g(\alpha^{-1} + 1_p)(\tau_i + 1_p) = \alpha \rho g(\rho_i + 1_p)(\alpha^{-1} + 1_p)$  ( $i \in [m]$ ), thus we obtain

$$\alpha \langle \pi_m^1 \rho g(\rho_1 + 1_p), \dots, \pi_m^m \rho g(\rho_m + 1_p) \rangle^+ = \tau(g(\rho + 1_p))^+$$

by an application of (F). Composing by  $\alpha^{-1}$  on the left will produce the result.

Summarizing the previous results we obtain:

**Theorem 1** An algebraic theory with iteration T is a generalized iterative theory if and only if each of the identities  $(A_1), (B_1), \dots, (F_1)$  is valid in T.

**Proof.** Sufficiency follows by the preceding lemmas. In order to prove necessity, we have to verify that  $(D_1)$  and  $(F_1)$  are valid in generalized iterative theories. But this is an immediate consequence of lemmas 2.5 and 2.3 in [2].

Next we introduce generalized scalar iterative theories. By scalar iteration we mean iteration restricted to scalar morphisms. That is, an algebraic theory with scalar iteration is an algebraic theory T, equipped with an operation  $^+$  which associates a morphism  $f^+ : 1 \rightarrow p$  with each scalar morphism  $f : 1 \rightarrow 1 + p$ . Generalized scalar iterative theories constitute an equational class of algebraic theories with scalar iteration. Namely, a generalized scalar iterative theory is an algebraic theory with scalar iteration which satisfies identities  $(A_1), (C_1), (D_1)$  as well as the following  $(E'_1)$  and  $(F'_1)$ :



$$(E'_1) \quad \varphi(\langle \pi_m^1 \rho g(\rho_1 + 1_p), \dots, \pi_m^m \rho g(\rho_m + 1_p) \rangle) = \varphi(g(\rho + 1_p))$$

where  $g: n \rightarrow m+p$  ( $n, m \geq 1, p \geq 0$ ),  $\rho: m \rightarrow n$  is a monotone base morphism,  $\rho_i: m \rightarrow m$  are base morphisms with  $\rho_i \rho = \rho$  ( $i \in [m]$ ). Furthermore, the term  $\varphi(f)$  where  $f$  is a variable of sort  $(n, p)$ , i.e.  $f: n \rightarrow p$ , is inductively defined as follows:

$$\varphi(f) = f^+ \quad \text{if} \quad n=1,$$

$$\varphi(f) = \varphi((1_{n-1} + 0_1) f < 1_{n-1} + 0_p, (\pi_n^n f \rho)^+, 0_{n-1} + 1_p >)$$

for  $n > 1$ , where  $\rho = < 0_1 + 1_{n-1}, 1_1 + 0_{n-1} > + 1_p$ .

$$(F'_1) \quad (f < 1_1 + 0_p, (g\rho)^+, 0_1 + 1_p >)^+ = f^+ < (g < f^+, 1_{1+p} >)^+, 1_p >,$$

where  $f, g: 1 \rightarrow 2+p$ ,  $\rho = < 0_1 + 1_1, 1_1 + 0_1 > + 1_p$ .

**Theorem 2** Every generalized iterative theory results in a generalized scalar iterative theory by restricting iteration to scalar morphisms. Conversely, every generalized, scalar iterative theory can be extended in a unique way to a generalized iterative theory.

Proof. The proof of the first sentence follows by observing that  $(E'_1)$  is just the combination of  $(E_1)$  with  $(B_1)$ , furthermore,  $(F'_1)$  can be deduced from identity  $(B_1)$  and its dual being also valid in every generalized iterative theory (cf. Lemma 2.4 in [2]).

Conversely, by identity  $(B_1)$  it is obvious, that there exists at most one appropriate extension for any generalized scalar iterative theory. Thus, we have only to show the existence of an extension. But this can be done immediately. Let  $T$  be an arbitrary generalized scalar iterative theory and extend operation  $^+$  by induction as follows:

$$0_p^+ = 0_p \quad (p \geq 0), \quad f^+ = \langle h^+, (\pi_n^n f \rho)^+ \langle h^+, 1_p \rangle \rangle \quad \text{if} \quad f: n \rightarrow n+p, \quad n \geq 1, \quad p \geq 0, \quad \text{where}$$

$$\rho = < 0_1 + 1_{n-1}, 1_1 + 0_{n-1} > + 1_p, \quad h = (1_{n-1} + 0_1) f < 1_{n-1} + 0_p, (\pi_n^n f \rho)^+, 0_{n-1} + 1_p >.$$



It can be seen that this definition of  $^+$  coincides with the original one on scalar morphisms, furthermore, in the theory obtained,  $(B_1)$  is obviously valid. By virtue of Theorem 1 and the fact  $(B_1), (E'_1) \models (E_1)$ , it is enough to verify that  $(F_1)$  is satisfied in  $T$ .

Let  $f, g: 1 \rightarrow 2+p$ ,  $h = f \langle 1_1 + 0_p, (g\rho)^+, 0_1 + 1_p \rangle$ ,  $k = g \langle f^+, 1_{1+p} \rangle$  where  $\rho = \alpha + 1_p$ ,  $\alpha = \langle 0_1 + 1_1, 1_1 + 0_1 \rangle$ . Observe that  $\rho^{-1} = \rho$ ,  $\alpha^{-1} = \alpha$ . Define  $\bar{f} = f\rho$ ,  $\bar{g} = g\rho$ ,  $\bar{h} = \bar{g} \langle 1_1 + 0_p, (\bar{f}\rho)^+, 0_1 + 1_p \rangle$ ,  $\bar{k} = \bar{f} \langle \bar{g}^+, 1_p \rangle$ . We know that  $h^+ = f^+ \langle k^+, 1_p \rangle$  and  $\bar{h}^+ = \bar{g}^+ \langle \bar{k}^+, 1_p \rangle$ . However,  $\bar{h} = g\rho \langle 1_1 + 0_p, f^+, 0_1 + 1_p \rangle = g \langle f^+, 1_{1+p} \rangle = k$ , and similarly,  $\bar{k} = h$ . Thus,  $k^+ = (g\rho)^+ \langle h^+, 1_p \rangle$  and we can compute in the following way:

$$\begin{aligned} (\alpha \langle f, g \rangle (\alpha^{-1} + 1_p))^+ &= \langle \bar{g}, \bar{f} \rangle^+ = \langle \bar{h}^+, (\bar{f}\rho)^+ \langle \bar{h}^+, 1_p \rangle \rangle = \\ &= \langle k^+, f^+ \langle k^+, 1_p \rangle \rangle = \langle (g\rho)^+ \langle h^+, 1_p \rangle, h^+ \rangle = \\ &= \alpha \langle h^+, (g\rho)^+ \langle h^+, 1_p \rangle \rangle = \alpha \langle f, g \rangle^+. \end{aligned}$$

### 3. TWO THEOREMS REGARDING THE CONNECTION BETWEEN ITERATIVE AND GENERALIZED ITERATIVE THEORIES

By a congruence relation of an iterative algebraic theory  $I$ , we mean an arbitrary congruence relation  $Q$  of  $I$  (being considered as an algebraic theory) and such that the quotient theory  $I/Q$  is an iterative theory. A congruence relation  $Q$  is called ideal if  $fQ\rho$  implies  $f=\rho$  for any  $f \in I$ ,  $\rho \in \theta$ . A congruence relation of a generalized iterative theory  $I$  is just a congruence relation of  $I$ , being considered as a many-sorted universal algebra. The congruence relations of a generalized iterative theory form a complete lattice. As it was pointed out in [2],  $I_\Sigma$ , i.e. the free iterative algebraic theory generated by a ranked alphabet  $\Sigma$  can be viewed as a weak subalgebra of  $R_\Sigma$ , the free generalized iterative theory generated by  $\Sigma$ . Thus, for any congruence relation  $Q$  of  $I_\Sigma$  there exists a smallest congruence



relation of  $R_\Sigma$  containing  $Q$ . This congruence relation is called the congruence relation generated by  $Q$ . The next lemma will give a simple description of this relation, provided  $Q$  is ideal.

Although, first we introduce a notation. For any congruence  $Q$  of  $I_\Sigma$  we shall denote by  $\bar{Q}$  the relation obtained by the following definition:

- (i) for  $f, g \in R_\Sigma(1, p)$ ,  $f \bar{Q} g$  if and only if there exist  $\tilde{f}, \tilde{g} \in I_\Sigma(1, p+1)$  such that  $f = \tilde{f}(1_p + 1)$ ,  $g = \tilde{g}(1_p + 1)$  and  $\tilde{f} Q \tilde{g}$ ,
- (ii) if  $f, g \in R_\Sigma(n, p)$ ,  $n \neq 1$ , then  $f \bar{Q} g$  if and only if  $\pi_n^i f \bar{Q} \pi_n^i g$  holds for any  $i \in [n]$ .

**Lemma 7** Let  $Q$  be an arbitrary ideal congruence relation of the iterative theory  $I_\Sigma$ . Then  $\bar{Q}$  is the congruence relation generated by  $Q$  in  $R_\Sigma$ . Furthermore,  $\bar{Q}|_{I_\Sigma} = Q$ .

**Proof.** By the properties of trees follows that  $\tilde{f}_1 = \tilde{f}_2$  is valid provided  $\tilde{f}_1(1_p + 1) = \tilde{f}_2(1_p + 1)$  and  $\tilde{f}_1, \tilde{f}_2 \in I_\Sigma(1, p+1)$ . As a consequence of this fact, one can easily verify that  $\bar{Q}$  is an equivalence relation. Also, using the tree concept, it can be seen that  $\tilde{f} \langle \tilde{g}_1, \dots, \tilde{g}_p, 0_q + 1_1 \rangle (1_q + 1) = f \langle g_1, \dots, g_p \rangle$  if  $f \in R_\Sigma(1, p)$ ,  $\tilde{f} \in I_\Sigma(1, p+1)$ ,  $g_i \in R_\Sigma(1, q)$ ,  $\tilde{g}_i \in I_\Sigma(1, q+1)$  ( $i \in [p]$ ), furthermore,  $\tilde{f}(1_p + 1) = f$  and  $\tilde{g}_i(1_q + 1) = g_i$  ( $i \in [p]$ ). Thus, if we have  $\tilde{f} Q \tilde{f}'$ ,  $\tilde{g}_i Q \tilde{g}'_i$  ( $i \in [p]$ ) as well as  $\tilde{f}'(1_p + 1) = f'$  and  $\tilde{g}'_i(1_q + 1) = g'_i$  ( $i \in [p]$ ), then  $f \langle g_1, \dots, g_p \rangle \bar{Q} f' \langle g'_1, \dots, g'_p \rangle$  follows by  $\tilde{f} \langle \tilde{g}_1, \dots, \tilde{g}_p, 0_q + 1_1 \rangle Q \tilde{f}' \langle \tilde{g}'_1, \dots, \tilde{g}'_p, 0_q + 1_1 \rangle$ . This, together with the second condition defining  $\bar{Q}$ , yields that  $\bar{Q}$  has the substitution property with respect to composition. The proof that  $\bar{Q}$  has the substitution property with respect to iteration is similar, only observe that  $\tilde{f}^+(1_p + 1) = f^+$  if  $\tilde{f}(1_{1+p} + 1) = f$  for  $f \in R_\Sigma(1, 1+p)$ ,  $\tilde{f} \in I_\Sigma(1, 1+p+1)$ , and use that  $\tilde{f}^+ Q \tilde{g}^+$  if  $\tilde{f} Q \tilde{g}$  and  $\tilde{f}$  is ideal, furthermore,  $\tilde{f} = \tilde{g}$  if  $\tilde{f} Q \tilde{g}$  and  $\tilde{f}$  is base.



Now we show that  $\bar{Q}|_{I_\Sigma} = Q$ . Assume that  $fQg$  is valid for  $f, g \in I_\Sigma(1, p)$ . Let  $\tilde{f} = f + 0_1$ ,  $\tilde{g} = g + 0_1$ . As  $Q$  is a congruence, we get  $\tilde{f}Q\tilde{g}$ . On the other hand we have  $\tilde{f}(1_p + 1) = f$  and  $\tilde{g}(1_p + 1) = g$ . This proves  $Q \subseteq \bar{Q}|_{I_\Sigma}$ . Conversely, supposing  $f\bar{Q}g$  for  $f, g \in I_\Sigma(1, p)$ , we obtain  $f + 0_1 Q g + 0_1$  since these are the only elements in  $I_\Sigma(1, p+1)$  with the properties  $(f + 0_1)(1_p + 1) = f$  and  $(g + 0_1)(1_p + 1) = g$ . However we have  $f = (f + 0_1) \langle 1_p, \pi_p^1 \rangle$  and  $g = (g + 0_1) \langle 1_p, \pi_p^1 \rangle$  if  $p > 0$ , and  $f = (f + 0_1)g$  and  $g = (g + 0_1)g$  if  $p = 0$ . In both cases  $f + 0_1 Q g + 0_1$  implies  $fQg$ .

Finally, we prove that if  $Q \subseteq Q'$  holds for a congruence relation  $Q'$  of  $R_\Sigma$ , then also  $\bar{Q} \subseteq Q'$ . Indeed, let  $f, g \in R_\Sigma(1, p)$  satisfy  $f\bar{Q}g$ . Then  $\tilde{f}Q\tilde{g}$  holds as well, where  $\tilde{f}, \tilde{g} \in I_\Sigma(1, p+1)$  are determined by  $f = \tilde{f}(1_p + 1)$  and  $g = \tilde{g}(1_p + 1)$ . As  $\tilde{f}Q\tilde{g}$  and  $Q \subseteq Q'$  we get  $\tilde{f}Q'\tilde{g}$  which in turn implies the required  $fQ'g$ .

In what follows, we shall prove that for every iterative theory  $I$  there exists a generalized iterative theory  $R$  together with an injective homomorphism (of partial algebraic theories with iteration), and such that for any homomorphism  $F: I \rightarrow T$  into a generalized iterative theory there is a unique homomorphism  $\bar{F}: R \rightarrow T$  with  $F = \eta \bar{F}$ , i.e.

**Theorem 3** Every iterative theory has a free extension in the class of all generalized iterative theories. Furthermore, this free extension is an iterative theory.

**Proof.** As each iterative theory is a homomorphic image of a free iterative algebraic under an ideal theory map, we may assume that  $I = I_\Sigma/Q$ , that is,  $I$  is the factor theory of the free theory  $I_\Sigma$  under an ideal congruence relation. As  $I_\Sigma$  has a free extension in the class of generalized iterative theories, namely,  $R_\Sigma$ , thus the correspondence  $\eta: I_\Sigma/Q \rightarrow R_\Sigma/\bar{Q}$  given by  $Q(f) \rightarrow \bar{Q}(f)$  ( $f \in I_\Sigma$ ) has the universal property. We have to show that  $\eta$  is injective. But this follows by  $\bar{Q}|_{I_\Sigma} = Q$ .



It remains to prove that  $R_{\Sigma}/\bar{Q}$  is iterative. By  $\bar{Q}|_{I_{\Sigma}} = Q$  it is nondegenerate. Now let  $f \in R_{\Sigma}(1, p)$ ,  $i \in [p]$ ,  $p \geq 1$ . Assume that  $f \bar{Q} \pi_p^i$ , and let  $\tilde{f} \in I_{\Sigma}(1, p+1)$  be the morphism determined by  $\tilde{f}(1_p + 1) = f$ . Then  $\tilde{f} Q \pi_{p+1}^i$  follows by  $\pi_{p+1}^i(1_p + 1) = \pi_p^i$ . As  $Q$  is ideal, this provides  $\tilde{f} = \pi_{p+1}^i$ . Therefore,  $f = \pi_p^i$ . We have seen that  $\bar{Q}$  is ideal. As both  $R_{\Sigma}$  and  $\bar{Q}$  are ideal, we get that  $R_{\Sigma}/\bar{Q}$  is also ideal. In order to verify the uniqueness of the iteration in  $R_{\Sigma}/\bar{Q}$ , by the main result in [1], it is enough to show that if  $f < g, 1_p > \bar{Q}g$  holds for some ideal  $f \in R_{\Sigma}(1, 1+p)$ ,  $g \in R_{\Sigma}(1, p)$ , then  $g \bar{Q} f^+$ . This property of  $\bar{Q}$  can be justified as follows. Let  $\tilde{f} \in I_{\Sigma}(1, 1+p+1)$  and  $\tilde{g} \in I_{\Sigma}(1, p+1)$  be those uniquely determined elements for which  $f = \tilde{f}(1_{1+p} + 1)$  and  $g = \tilde{g}(1_p + 1)$  hold. Then also  $\tilde{f} < \tilde{g}, 1_{p+1} > \cdot (1_p + 1) = f < g, 1_p >$  and  $\tilde{f}^+(1_p + 1) = f^+$  are valid. Hence,  $\tilde{f} < \tilde{g}, 1_{p+1} > \bar{Q}\tilde{g}$  which by the uniqueness of the iteration in  $I_{\Sigma}/Q$  implies that  $\tilde{g} Q \tilde{f}^+$ . Therefore,  $g = \tilde{g}(1_p + 1) \bar{Q} \tilde{f}^+(1_p + 1) = f^+$ .

**Theorem 4** Let  $I$  be an iterative theory and  $\Omega \in I(1, 0)$ . There exists a unique way to extend the operation  $^+$  such that  $I$  becomes a generalized iterative theory with  $\Omega = \pi_1^{1+}$ .

**Proof.** Again, we may assume that  $I = I_{\Sigma}/Q$  for a ranked alphabet  $\Sigma$  and ideal congruence  $Q$ . Denote by  $Q_1$  the congruence relation of  $R_{\Sigma}$  generated by  $\{(1, \Omega)\}$ . By Theorem 3, we know that the mapping  $F: Q(f) \rightarrow (Q_1 \vee \bar{Q})(f)$  is a homomorphism of  $I_{\Sigma}/Q$  into  $R_{\Sigma}/Q_1 \vee \bar{Q}$ . Here  $Q_1 \vee \bar{Q}$  denotes the l.u.b. of  $Q_1$  and  $\bar{Q}$  in the congruence lattice of  $R_{\Sigma}$ . As  $\Sigma$  generates  $R_{\Sigma}$  and  $\Omega Q_1^1$ , we obtain that  $F$  is surjective. We shall prove that  $F$  is even injective.

Assume that  $f Q_1 \vee \bar{Q} g$  holds for  $f, g \in I_{\Sigma}(1, p)$ . Then there exist morphisms  $f_0, f_1, \dots, f_{2n} \in R_{\Sigma}(1, p)$  ( $n \geq 0$ ) with  $f = f_0 Q_1 f_1 \bar{Q} f_2 \dots$

$\dots f_{2n-1} \bar{Q} f_{2n} = g$ . Let  $\tilde{f}_0, \tilde{f}_1, \dots, \tilde{f}_{2n} \in I_{\Sigma}(1, p+1)$  be uniquely determined by  $\tilde{f}_i(1_p + 1) = f_i$  ( $i \in \{0, \dots, 2n\}$ ). Define  $f'_i$  by  $f'_i = \tilde{f}_i(1_p + \Omega)$ . Of course, we have  $f_0 = f'_0$  and  $f_{2n} = f'_{2n}$ . By the definition of  $\bar{Q}$ ,  $f'_i \bar{Q} f'_{i+1}$  holds for every  $i$  is odd. On



the other hand  $f'_i Q_1 f'_{i+1}$  holds for every even  $i$ . Thus,  
 $f = f'_0 Q_1 f'_1 \bar{Q} f'_2 \dots f'_{2n-1} \bar{Q} f'_{2n} = g$ . However,  $Q_1|_{I_\Sigma}$  is the equality  
 relation and  $\bar{Q}|_{I_\Sigma} = Q$ . Therefore,  $fQg$  can be derived.

We have shown that  $F$  is a bijective homomorphism. This  
 results that the definition  $f^+ = F^{-1}(F(f)^+)$  ( $f \in I_{\Sigma/Q}$ ) gives a  
 satisfactory extension to a generalized iterative theory. The  
 uniqueness of the extension is obvious.

#### REFERENCES

- [1] Bloom, S.L, S. Ginaly and J.D. Rutledge, Scalar and  
 vector iteration, JCSS 14(1977), 251-256.
- [2] Ésik, Z., Identities in iterative and rational algebraic  
 theories, CL and CL (Computational Linguistics and  
 Computer Languages XIV. 1980.)



## A VERY HIGH LEVEL DISCRETE SIMULATION SYSTEM T-PROLOG

*Iván Futó and János Szeredi*

Institute for Coordination of  
Computer Techniques  
Budapest, Hungary

### ABSTRACT

T-PROLOG a very high level simulation language is presented. It has the following properties.

- the system takes over part of the problem solving effort from the user,
- a built-in backtrack mechanism permits backtracking in time in case of a deadlock.
- it changes automatically and dynamically the simulation model on the basis of logical consequences.
- a more advanced process communication mechanism is presented for the user.

The processes are synchronized by a built-in scheduler.



## 1. INTRODUCTION

Solving problems using computerised models of real systems is one of the purposes of AI and simulation research. The developed programming tools, AI [1], [3], [4], [5], [7], and simulation [2], [8], [9], [10] languages differ in many features. We found it useful to construct a modelling system which takes the time handling concepts from the simulation languages and the non-algorithmic programming concept, pattern matching and backtracking from the AI languages.

Our system is implemented in PROLOG [1], [6], [7], a logic based AI language with simple syntax (Horn clauses) and great expressive power due to its logical basis. It has a mechanical theorem-prover with depth-first strategy. Some features of the experimental simulation system called T-PROLOG is presented in the following paragraphs.



## 2. INTRODUCTION TO VERY HIGH-LEVEL COMPUTER SIMULATION

### 2.1 SIMULATION USING DIGITAL COMPUTERS

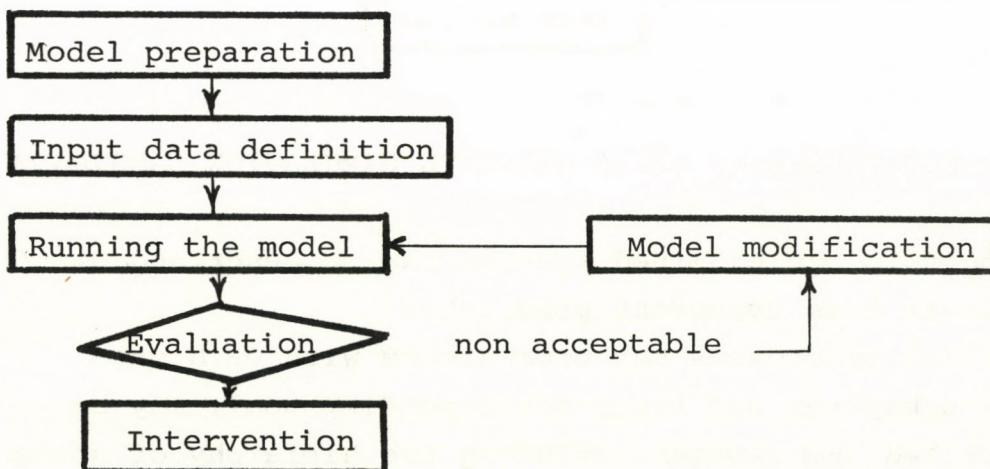
By simulation we mean discrete event simulation, when changes occur in discrete time moments.

Simulation is a problem solving procedure. We prepare a computerised model (a program) of real life system which reflects the interesting characteristics to examine the behaviour of the original system. (i.e. by running the program).

The purpose of the simulation can be

1. The examination of the reaction of the observed system when its structure or parameters are changed.
2. To examine the final state the system reaches from a given initial one and how its structure or parameters have to be changed to obtain a predefined final state. We will call this kind of simulation goal oriented.

The shema which corresponds to the goal-oriented simulation is the following:

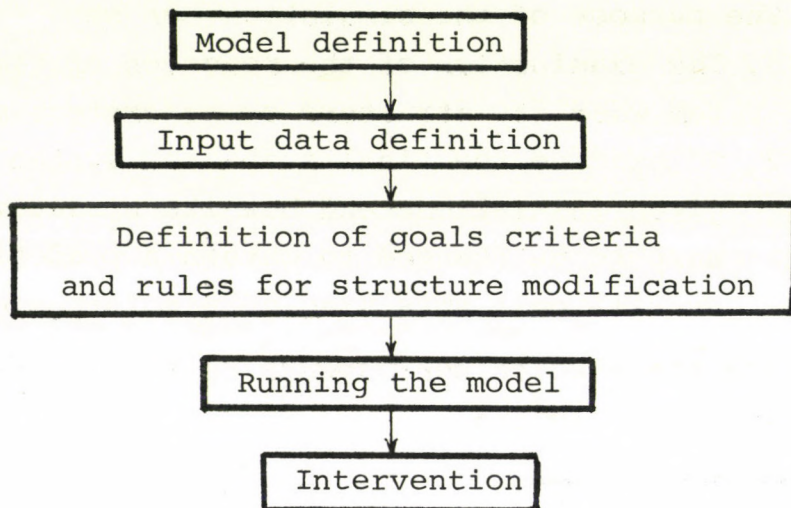


In the above schema the meaning of intervention is an action of the modeller on the real system to achieve the wanted results. Evaluation and modification of the model is a human activity which follows each running of the program.



## 2.2 VERY HIGH LEVEL COMPUTER SIMULATION

We call the simulation procedure very-high level simulation if the behaviour of the model is tested on different points during the running of the program by control conditions and when the conditions or the prescribed goals cannot be fulfilled, back-track occurs. In this case the structure of the original model can be changed automatically according to different predefined rules depending on the actual state of the system. Then the original schema of the simulation procedure can be modified in the following way:



## 2.3 CRITERIA FOR VERY-HIGH LEVEL DISCRETE SIMULATION LANGUAGES

1. Changes in the examined systems can be described by explicit time dependent predicates
2. Parallel processes may communicate with each other.
3. Each subsystem and their corresponding goal, may be described and examined including the hierarchy of goals determined by the relations between the subsystems.

From the system simulation's point of view, the role of time is essential. As real systems work in time, the conditions for synchronising their processes are time dependent.



We want to satisfy the above conditions in the following ways:

1. By using a logic-based very high-level language which is suitable for the simultaneous description of the processes and goals of the systems, and is interactive, easy to modify and modular.
2. During the simulation run, the program system assigns to each process a theorem prover whose task is to prove the goal of the process by means of the logical statements defining that process.
3. To solve the conflicts which arise from the possibility of data base modifications, or resource allocation sophisticated rules of preferences can be given.
4. Back-tracking is provided to some previous state from which an alternate path can be chosen to try to go forward again toward the goal. This is important in dead-lock situations, from which goals cannot be reached.
5. The set of formulas, defining the simulation model can be altered dynamically during the running of the program. This means that the original structure of the simulation model can be modified automatically using predefined state dependent rules.
6. During program execution new processes with new goals can be created and old ones can be deleted from the simulation model. In the case of deletion the successful proof of the corresponding goal is not considered.

T-PROLOG is the first experimental realisation of such a simulation system.

For traditional simulation problems T-PROLOG works as a process oriented simulation system.



### 3. BASIC ELEMENTS OF T-PROLOG

The first basic feature of the language is the process. Termination of a T-PROLOG program means the successful termination of some processes. The processes can be created at the initial phase of the simulation as well as during runtime. Processes conceptually run parallel in time. A goal to be achieved corresponds to each process.

The second basic notion of T-PROLOG is the resource. Resources are model elements that can be used only by one process at time. The use of a resource can take time. Depending on the type of the resource after of its use it becomes available or not for other processes. Resources may be created at program initialisation or dynamically during program execution.

The third basic feature in T-PROLOG is the notion of the internal time which is maintained by an internal clock. To every process which is not waiting for the fulfilment of a condition, a time moment is assigned, when its execution will continue. The execution of a process can be suspended for a time interval "T", in this case its reactivation time will be the actual time (told by the internal clock) + "T".

To synchronise processes, and to control time, different built-in predicates are used. With the help of these predicates the processes can communicate with each other in three ways:

1. By using common logical variables.

A process is suspended until another process assigns an appropriate value to their common variable.

2. By using the common shared database.

The execution of a process is suspended until another process adds the needed assertion to the database.



### 3. By sending messages.

The execution of a process is suspended until another process sends the message it is waiting for.



## 4. PROGRAM EXECUTION

### 4.1 PRELIMINARY NOTIONS

To understand the execution of a T-PROLOG program we need the definition of the following process classes.

1. The currently executed process is the active process. (In a one processor implementation only one process can be active).
2. Those processes, which are suspended because of time consideration or which are not yet started, are waiting processes. To each such process a time "t" is assigned, which indicates when the process execution has to continue. These are the processes which form the waiting list. This list is ordered according to "t".
3. Processes which are waiting for some message are the "demon" processes, they form the demon list. If a process on this list receives the wanted message, then with the actual system time told by the internal clock as reactivation time, it is put to the head of the waiting list.
4. Those processes which are suspended because of a non-time dependent condition, are the blocked processes, which are placed on the blocked list.

We need also the definition of the activable process. At time "t" the following processes are activable:

1. - Those processes on the waiting list whose activation (reactivation) time is "t" and
- Those processes on the blocked list whose blocking conditions are fulfilled in this time moment.



2. If there is no such process and the activation (reactivation) time of the first process on the waiting list is "t1" ( $t < t1$ ), then those elements on the waiting list are activable whose activation time is "t1".

If the current active process is suspended then an activable process gets control. If there is no such process then back-track begins.

In case 2. The internal clock moves from "t" to "t1".

#### 4.2 BUILT-IN CONTROL PREDICATES

The control predicates are those built-in predicates which modify the spelling of the original PROLOG interpreter.

1. Wait (g)

Syntactical restrictions: "g" atom

This predicate suspends processes depending on logical conditions. "g" is the condition for suspension.

2. Wait for (a), Send (a)

Syntactical restrictions: "a" term

Predicates to send and wait for messages.

3. New (gl,x), New(gl,x,st), New(gl,x,st,et)

Syntactical restrictions: "gl" list of atoms (goals), "x" is a term "st" and "et" are integers.

These predicates create processes. "gl" is the goal-sequence of the process, "x" is its identifier "st" is the starting time and "et" is the prescribed time to achieve the goal-sequence. The default values are:

"st"=system time, "et"=100000.



4. Hold(t), During(t)

Syntactical restrictions: t integer

Unconditionally suspends processes for "t" time units.

5. Delete(p)

Syntactical restriction: p term

Those processes whose identifier match "p" will be deleted from the system.

6. Seize (r), Release(r)

Syntactical restrictions: "r" term

These predicates seize and release resource of processes.

7. Addresource(r,n)

Syntactical restrictions: "r", term "n" pos. integer. This predicate adds dynamically "n" instances of type "r" of resources to the model.

#### 4.3 PREDICATES FOR DATABASE/PROGRAM MODIFICATION

1. Add(cl,n)

Syntactical restrictions: cl list of literals, n is an integer. This predicate serves to insert into the database a clause, whose literals are the members of cl, to the n-th position of the corresponding partition.

2. Del(id,n)

Syntactical restrictions: id is an identifier, n is an integer. The predicate serves to delete from the i-th partition the n-th clause.



#### 4.4 THE ALGORITHM OF THE PROGRAM EXECUTION

We shall use the following abbreviations:

Active process	AP
Waiting list	WL
Blocked list	BL
"Demon" list	DL
System time	T
Seize List	SL is formed from pairs (p:r) where p is the identifier of a process waiting for the resource r.
Resource list	RL RL is the list composed from the pairs (r:n), "r" being a resource type, "n" the number of available resources of type "r".

AP is always a member of WL or BL.

##### 1. Program initialisation

The goal-sequence of the simulation model consists of "k" process initialisations:

:New(g<sub>1</sub>,n<sub>1</sub>,ts<sub>1</sub>,tel),...,New(g<sub>k</sub>,n<sub>k</sub>,ts<sub>k</sub>,tek).

"default values for "tsi" and "tei" are  
0 and 100000.

Now we define the waiting list, blocked list,  
"demon" list, active process and the system time.

WL<sub>1</sub>=(n<sub>1</sub>:ts<sub>1</sub>).(n<sub>1</sub>:ts<sub>2</sub>). ... .(n<sub>k</sub>:ts<sub>k</sub>). Nil

WL = WL<sub>1</sub> ordered by "ts" in monoton increasing  
way

BL = empty, DL:=empty

if the first element of W<sub>1</sub> is n:ts. then

AP:=n and T:=ts



On the WL list the goal-sequence of process "ni" is "gi" "tsi" is the prescribed activation time and "tei" is the termination time.

## 2. The execution of the active process

The goal-sequence of process AP is executed as in PROLOG until the goal-sequence becomes empty or the first goal of the sequence is a control built-in predicate.

## 3. Interruption of process execution

There are two possibilities:

(1) The goal-sequence of the process is Nil. Then AP is deleted from WL or BL and the program execution continues at 4.

(2) The first goal of the goal-sequence of AP is the control predicate "C".

a. C=Wait(g),

if "g" can be proved then the execution of "C" succeeds by proving "g" and the execution of AP continues at 2. Otherwise (if "g" could not be proved AP) is deleted from WL or BL (it depends on which list contains AP). We put AP at the end of BL, the condition of suspension is "g".

b. C=Wait\_for(m)

AP is deleted from BL or WL. DL:=DL appended AP to the end. The "demon" condition of AP is "m". Program execution continues at 4.

c. C=Send(m)

Those elements of DL, whose "demon" condition can be



unified with "m" are inserted into the head of list WL. Program execution continues at 4. If there is not such process on the "demon" list then back-track begins.

d.  $C = \text{New}(gl, x)$ ,  $C = \text{New}(gl, x, st)$ ,  $C = \text{New}(gl, x, st, et)$

The default value of "st" is  $st := T$  the actual system time, and of "et" is  $et := 100000$ .

If  $st < t$  then back-track occurs. Otherwise  $WL := WL$ , with insertion "x" activation time "st". If in WL there are already processes with activation (reactivation) time "st" then "x" is inserted as the latest one. The goal-sequence of "x" is "g", (The time to terminate is "et". The program execution continues at 2. The execution of the active process continues.)

e.  $C = \text{Hold}(h)$ ,  $C = \text{During}(h)$

If  $h < 0$  then back-track begins. If  $ts = T + h$  is greater than the prescribed time for completion of AP then back-track occurs too. Otherwise AP is deleted from WL or BL and is inserted into WL with a reactivation time of  $ts = T + h$ . The program execution continues at 4.

f.  $C = \text{Delete}(p)$

Those processes whose identifier can be matched with "p" are deleted from WL, BL, and DL except AP.

(so, if "p" is a variable then all processes are deleted except AP.) The program execution continues at 2. with AP.

g.  $C = \text{Seize}(r)$

There are two possibilities: for some n,  $(r:n)$  is in RL or not. If  $(r:n)$  is not an element of RL, then the system sends an error message and the program execution terminates. If  $(r:n)$  is a member of RL. Then AP is deleted from VL or BL.  $AP:r$  Appended to the end of SL, program execution continues at 4.



h. C=Release(r)

There are two cases too. If AP does not have a resource of type r (it has not seized or has already released the resource) then the system sends an error message and the program execution is terminated. If AP seized r then in RL (r:n) is changed to (r:n+1) and AP continues its run at 2.

i. C=Addresource (r,n)

If (r:m) is an element of RL then it is changed to (r:m+n), if it is not the case then (r:n) is appended to the end of RL.

4.

- (1) If WL=DL=BL=empty then the program execution has terminated successfully.

Otherwise:

- (2) If the activation time of the first element of WL is equal to  $T_s$  (the actual system time) then this process will be the new AP and program execution continues at 2.

Otherwise:

- (3) If there is a process in the BL, whose suspension condition is fulfilled then AP will be the first from these processes. The program execution continues at 2.

Otherwise:

- (4) Definition: homogenous process class.

Two processes are homogenous from the point of view of a resource, if at the moment when they want to seize the resource all their parameters, except the process identifier, are the same. Homogenous processes form a homogenous process class. In case of back-track, permutation of processes to receive a resource is executed only for

the elements of different homogenous process classes. No processes belonging to the same class are permuted.

More formally:

If in SL, there is no such  $(p:r)$  for which  $(r:n)$  in RL,  $n>0$ , then the control omits this point.

Otherwise:

Let  $r_1, r_2, \dots, r_k$  be resources for which  $r_i:n_i$  is an element of RL, and  $n_i>0$ , and for  $r_i$  there is such  $p_i$  that  $p_i:r_i$  is an element of SL.

If  $p_{i1}, \dots, p_{iil}$  are those process identifiers for which  $p_{ij}:r_i$  are members of SL (the order of these depends from the order of RL and SL).

Let us rename  $p_{ij}$ -s as follows:

$q_1, q_2, q_3, \dots, q_n$

$p_{11}, \dots, p_{1il}, p_{21}, \dots, p_{2i2}, p_{31}, \dots, p_{k1}, \dots, p_{kik}$

( $q_1=p_{11}$  and  $q_n=p_{kik}$ ) and execute the following loop:

$m:=1$

a.  $AP:=qm$ , AP is appended to VL with start time  $t$ ,  $qm$  is the new name of  $p_{ij}$ . Therefore on RL we change  $(r_i:m_i)$  to  $(r_i:m_i-1)$ . And program execution continues at 2. If during backtrack control reaches a., then execution continues at b.

b. If  $m=n$  then backtrack continues.

If  $m<n$  and  $q_s, s>m$  is the identifier of the first process which does not belong to the same homogenous process class as  $qm$  then  $m:=s$  and execute a.

If there is no such  $q_s$ , then backtarck continues.

(5) If WL is not empty, its first element is " $n:ts$ " and the termination time of all processes is less than or equal to " $ts$ ". Then  $AP=n$  and  $t:=ts$  and program execution continues at 2.

Otherwise:



(6) Back-track

That means that back-track occurs when there is no element of BL whose suspension condition is fulfilled and WL is empty or the activation time of its first element is greater than the prescribed termination time of another process.

5. Simple T-PROLOG example for resource distribution

The problem is a variant of the well-known five-philosopher problem of Brinch Hansen.

Five philosophers sit at a round table. There are five forks on the table. Every philosopher needs 2 forks for eating but they may use only the forks at their right side and left side. We have to have them finish their meal in a non trivial way, that is we prescribe their lunch time short enough not to allow the trivial solution - the philosophers can't have their meal simply one after another.

Let the lunch time be 30 minutes and let's suppose that every philosopher is able to finish his own portion within 10 minutes. The corresponding T-PROLOG program is the following.

```
Resource(Fork(A),1).  
Resource(Fork(B),2).  
Resource(Fork(C),1).  
Resource(Fork(D),1).  
Resource(Fork(E),1).
```

Resource declarations.

```
Satiated(n):  
Good-fork(n,f1,f2),  
Seize(Fork(f1)),  
Seize(Fork(f2)),  
Eat,  
Release(fork(f1)).  
Release(fork(f2)).  
Eat:
```

During(10).

```
Good-fork(1,A,B).  
Good-fork(2,B,C).  
Good-fork(3,C,D).  
Good-fork(4,D,E).  
Good-fork(5,E,A).  
End.
```

```
:New(Satiated(1).Nil, F1,0,30), New(Satiated(2). Nil, F2,0,30),  
New(Satiated(3).Nil, F3,0,30), New(Satiated(4). Nil, F4,0,30),  
New(Satiated(5).Nil, F5,0,30).
```

The main steps of the program execution are:

1. 1-st Philopsher enters in the waiting queue of the resource representing fork A. The 2-Nd philosopher enters in the queue of fork B. Similarly the other philosophers enter in queues of forks C, D, E respectively.
2. 1-st philosopher receives fork A leaves the queue of fork A and enters in the queue of fork B. He is the second member of the queue, philosopher 2 stays before him. Then 2-nd



philosopher receives fork B, leaves queue of fork B and enters in the queue of fork C. He will be the second member of this queue philosopher 3 is staying before him. So do philosophers 3, 4 and 5.

3. Then every philosopher has one fork and waits for the other one but these forks are in the possession of an other philosopher. This is a typical deadlock situation.
4. Backtrack begins. There was a previous system state when in the queue of fork E there were philosophers 5 and 4, in this order, philosopher 4 having the fork D. The two philosophers belong to different homogenous process class respecting fork E. So they can be permuted during backtrack and now philosopher 4 receives fork E, (Previously philosopher 5 received it.) Philosopher 4 now has his two forks D and E while philosopher 5 has no fork at all and remains in the queue of fork E.
5. In the system at this moment only philosopher 4 can eat. System time becomes 10 which means that philosopher 4 finished the eating, he releases forks E and D.
6. Philosophers 3 and 5, who wait for these forks now can seize them. Now philosophers 1 and 2 have to wait for their missing forks (B and C), While philosopher 5 waits for fork A. Then system time is set to 20. Philosopher 3 finishes his meal and releases fork C and D. This means that now philosopher 2 who waited for fork C can eat, but the other philosophers (1 and 5) can not.
7. System time is set to 30 philosopher 2 releases forks B and C. Philosopher 1 immediately seizes fork B but the T-PROLOG scheduler finds that he can finish his meal only for 40 and the permitted limit is 30 so a new backtrack begins.



8. Backtrack continues until such a resource (fork) distribution is not found that at time moment 0 2-nd philosopher receives forks B and C, philosopher 4 receives forks D and E. Then for time 10 they can finish eating and philosopher 1 and 3 can receive their missing forks. They finish eating at 20 and release the forks and finally philosopher 5 can eat too, and he finishes for 30.

9. The above solution is the first one that the system can find.

## 6. Conclusion

T-PROLOG, a very high level simulation language has the following advantages compared to the traditional simulation languages:

1. The system assumes over some of the problem solving effort from the user who then is free to concentrate on defining the task rather than solving it;
2. The system changes the model automatically and dynamically on the base of logical consequences derived from sophisticated preconditions;
3. A built in back-track mechanism permits backtracking in time in case of deadlock or hopeless intermediate situation (a logical condition which is necessary for the continuation of the execution is missing, or the current time condition has become contradictory);
- 4..T-PROLOG has a process communication mechanism which includes interprocess communication through variables which are evaluate by pattern matching or by modification of the model description. (We preserve the traditional way of communication too: the processes are able to send and receive messages.).



## 7. Plans for the future

Our hope is to transform the current version of T-PROLOG to a simulation system that for traditional discrete simulation problems runs as effectively as the traditional simulation languages. [2], [9], [10]

An other task to solve is to find more "intelligent" backtracking strategies, automatic or user controlled.

An interactive program development and debugging subsystem is in its planning phase. Other facilities to help interactive simulation, giving the possibility to interrupt program execution, modify the model from user terminals, force the system to back-track to a given state are under planning.

The experimental version of T-PROLOG is used to model regional developments.

## REFERENCES

- 1/ G.Battani.H.Meloni  
Interpreter du Language de Programation PROLOG  
Universite D'Ais Marseille sept. 1973
- 2/ GPSS/360 User's Manual  
IBM 420-03261
- 3/ C.Hewitt  
PLANNER: A language for Manipulating Models and Proving  
Theorem in a Robot  
Int. Joint. Conf. Artificial Intelligence 1969.  
Washington DC.
- 4/ C.Hewitt  
Viewing Control Structures as Patterns of Passing Messages  
Artificial Intelligence Vol 8. No.3. june 1977

- 5/ MC Dermott, G.J.Sussman  
The CONIVER Reference Manual  
MIT AT Laboratory Memo no. 259, Cambridge may 1972
- 6/ R. Kowalski  
Logic for Problem solving  
North-Holland 1980
- 7/ MC. Cabe  
Programmer's Guide to IC-PROLOG  
Imperial College London 1978
- 8/ J.F. Rulifson, J.A. Derksen, R.J. Waldinger  
QA4 A Procedural Calculus for Reasoning  
AT Center Technical Note SRT Project 8721, 1972
- 9/ P.J. Kiviat  
Simulation Using Simscript II.  
Rand Corporation 1968
- 10/ Simula Reference Manual  
Control Data 6400/6500/6600 Computer Systems
- 11/ S.Wendt  
BORIS a Block Oriented Interactive Simulation System  
University of Kaiserslautern W.Germany 1978
- 12/ I.Futó, J.Szeredi  
T-PROLOG General Information Manual  
Institute for Coordination of Computer Techniques  
1981.





## A PARSING METHOD BASED ON VAN WIJNGAARDEN GRAMMARS

László Gerevich

Computer Centre of the National Planning Office  
Budapest, Hungary

### Abstract

The following is to demonstrate a top-down parsing method based on van Wijngaarden grammars ("W-grammars" for short; also called two-level grammars), the application of which does not require any radical change in the given W-grammar, or if certain stipulations are taken into consideration in the defining process, none at all.

The parsing method is applicable

- to check the W-grammar, describing the context-free syntax and context conditions of the language in question, in the stage of development;
- to a compiler of the language, in its original general form, used in the checking process, and also in an optimized version.

### 1. Introduction

Our method is described in three steps. *First*, a modified definition of the derivation according to W-grammars is given, which however retains the generated language. The modification renders the W-grammars more suitable for parsing purposes, independent of the actual parsing purposes, and independent of the actual parsing method employed. The *second* step is the description of an extension of the chosen method, that of recursive descent. The new elements introduced are the grammar-type parameters and the search for rules which make the method of recursive descent suitable for designing parsers operating on the basis of W-grammars. *Third* comes the description of the implemented method, being an extended version of Knuth's parsing machine [12], the instructions of which are executed through macrocommands. D.A. Watt in [20] was the first to apply W-grammars in parsers, by introducing the grammar-type variables into affix-grammars [13], in addition to affixes (EAG). This EAG was realized, as well as further developed (ECDL) by B. Kramer and H.W. Schmidt [15],[16].

The present method, unlike ECDL, does not require essential changes in W-grammars, since it allows

- the application of general hyperrules, as e.g. predicates;
- a more flexible use of grammar-type parameters.

In this paper we give a description of the first step and, briefly, of the second step too. In forthcoming papers we intend to describe the restrictions on W-grammars in a more



detailed manner, ensuring that parsers made by these grammars work and recognize the languages generated by them; furthermore we also wish to demonstrate the use of this method in practice.

## 2. Van Wijngaarden Grammars

The formal definition of W-grammars introduced by J.L. Baker [1] and developed by P.A. Deussen [5] is recalled and demonstrated on a simple example. It is followed by the informal description of the modified derivation according to the W-grammar and its formal definition. The equivalency of these two definitions is proved.

### 2.1. Formal definition of W-grammars

**Definition 1.** A W-grammar is an ordered septuple

$$WG = (M, \tau, \Pi, \Sigma, \Phi, \Gamma, \langle s \rangle),$$

where

- $M$  — a finite set of metanotions,
- $\tau$  — a finite set of metaterminals ( $M \cap \tau = \emptyset$ )
- $\Pi \subset M \times (M \cup \tau)^*$  — a finite set of context-free metaproductions (metarules), the quadruple  $(M, \tau, \Pi, A)$  is a context-free grammar for each metanotion  $A \in M$ , the sets  $M$ ,  $\tau$  and  $\Pi$  are sometimes called metasystem or metalevel;
- $\Sigma$  — a finite set of terminals of the language,
- $\bar{\Phi} \subset \{ \langle u \rangle : u \in (M \cup \tau)^+ \}$  — a finite set of hypernotations,
- $\bar{\Gamma} \subset \bar{\Phi} \times (\bar{\Phi} \cup \Sigma)^*$  — a finite set of hyperrules that serve as schemata for productions,
- $\langle s \rangle \in \bar{\Phi}$  — a distinguished hypernotation called starting symbol.

The sets  $\Sigma$ ,  $\bar{\Phi}$  and  $\bar{\Gamma}$  are called language level.

**Definition 2.** The semigroup homomorphism  $\Psi$

$$\Psi : (M \cup \tau \cup \Sigma \cup \{ <, >, \rightarrow \})^+ \rightarrow (\tau \cup \Sigma \cup \{ <, >, \rightarrow \})^*$$

is called *consistent replacement* (universal assignment or uniform replacement rule), if and only if (iff) the following condition is satisfied

$$\Psi(X) = \begin{cases} u \in L_X, & \text{if } X \in M \text{ (} L_X = \{ v \in \tau^*, X \xRightarrow{\Pi} v \}) \\ X, & \text{if } X \in (\tau \cup \Sigma \cup \{ <, >, \rightarrow \}) \end{cases}$$

The possibly infinite set  $\Gamma$  of productions for the language obtained by consistent replacements from the hyperrules is  $\Gamma = \{\gamma : \bar{\gamma} \in \bar{\Gamma}, \gamma = \Psi(\bar{\gamma})\}$ .

The items of the set are called notions  $\Phi = \{\langle u \rangle : u \in \tau^* \text{ and there are a hyperrule } \gamma : \langle w \rangle \rightarrow \xi \in \Gamma \text{ and consistent replacement } \Psi \text{ for which } \Psi(w) = u\}$

The string of notions and terminals of the language derived from the starting symbol by the productions of  $\Gamma$  is called *sentential form*  $L_{WG} = \{\mu : \langle s \rangle \xRightarrow{\Gamma} \mu\}$ . The language generated according to the W-grammar is the set of sentential forms containing only terminals of the language

$$L_{WG} = L_{WG} \cap \Sigma^* = \{\sigma \in \Sigma^* : \langle s \rangle \xRightarrow{\Gamma} \sigma\}$$

**Example 1.** The grammar generating the language  $\alpha^k \beta^k \alpha^k$  is

$$WG_1 = (M_1, \tau_1, \Pi_1, \Sigma_1, \Phi_1, \Gamma_1, \langle s \rangle)$$

where

$$M_1 = \{N\}, \tau_1 = \{a, n\}, \Pi_1 = \{N \rightarrow nN, N \rightarrow \lambda \text{ (empty string)}\}, \Sigma_1 = \{\alpha, \beta\},$$

$$\Phi_1 = \{\langle s \rangle, \langle aN \rangle, \langle bN \rangle, \langle anN \rangle, \langle bnN \rangle, \langle a \rangle, \langle b \rangle\},$$

$$\Gamma_1 = \{\gamma_1 : \langle s \rangle \rightarrow \langle aN \rangle \langle bN \rangle \langle aN \rangle$$

$$\gamma_2 : \langle anN \rangle \rightarrow \alpha \langle aN \rangle$$

$$\gamma_3 : \langle bnN \rangle \rightarrow \beta \langle bN \rangle$$

$$\gamma_4 : \langle a \rangle \rightarrow \lambda$$

$$\gamma_5 : \langle b \rangle \rightarrow \lambda\}$$

The word  $\alpha\beta\alpha$  is generated by the context-free grammar  $G_1$  derived from the hyperrules  $G_1 = (\{\langle an \rangle, \langle bn \rangle, \langle a \rangle, \langle b \rangle, \langle s \rangle\}, \{\alpha, \beta\},$

$$\{r_1 : \langle s \rangle \rightarrow \langle an \rangle \langle bn \rangle \langle an \rangle, r_2 : \langle an \rangle \rightarrow \alpha \langle a \rangle, r_3 : \langle bn \rangle \rightarrow \beta \langle b \rangle, r_4 : \langle a \rangle \rightarrow \lambda,$$

$$r_5 : \langle b \rangle \rightarrow \lambda\}, \langle s \rangle); \Psi_1(\gamma_1) = r_1 (\Psi_1(N) = n), \Psi_2(\gamma_2) = r_2 (\Psi_2(N) = \lambda)$$

$$\Psi_2(\gamma_3) = \gamma_3 = r_3$$

The hyperrules  $\gamma_4$  and  $\gamma_5$  are productions of the language as they have no metations. The derivation of the word  $\alpha\beta\alpha$  by  $G_2$  is:

$$\begin{aligned} \langle s \rangle &\rightarrow \langle an \rangle \langle bn \rangle \langle an \rangle \rightarrow \alpha \langle a \rangle \langle bn \rangle \langle an \rangle \rightarrow \alpha \langle bn \rangle \langle an \rangle \rightarrow \alpha \beta \langle b \rangle \langle an \rangle \rightarrow \alpha \beta \langle an \rangle \rightarrow \\ &\rightarrow \alpha \beta \langle a \rangle \rightarrow \alpha \beta \alpha \end{aligned}$$



Note: two notions are equal iff their metaterminal representations (metaterminal strings) are equal.

The (infinite) language is generated by (possibly infinite) context-free grammars obtained from the finite set of hyperrules by consistent replacement, defined by metaproducts. In the case of grammar  $WG_1$ , the application of a consistent replacement ( $\Psi_1$  in the derivation in Example 1) to obtain the production  $r_1$  of the language, from the hyperrule  $\gamma_1$ , was determinant from the aspect of the generated word  $\alpha\beta\alpha$  as the number  $k(1)$  of the symbols  $\alpha$ ,  $\beta$  and  $\alpha$  again equals the number of the metaterminal symbols  $n(1)$  in the string, replacing metanotation  $N$  in the hyperrule  $\gamma_1$ . The actual relationship between the generated word  $\alpha\beta\alpha$  and the determinant consistent replacement  $\Psi_1(N) = n$  depends on the other hyperrules (language level). If the hyperrule  $\gamma_2$  were  $\langle annN \rangle \rightarrow \alpha \langle aN \rangle$ , the relationship would be different.

## 2.2. Modified definition of the derivation according to the W-grammar

The separation of the two levels presents a problem which we propose to solve by modifying the definition of the derivation according to the W-grammar. The essence of this modification consists in preserving the metanotations in the steps of derivation until the application of a hyperrule-production does not call for their replacement (postponed metaderivation). The notion of consistent replacement is extended in such a way that the metanotations are replaced by strings of metasymbols (not only metaterminals) in each of their occurrences. This replacement is called *extended consistent replacement*.

The application of a derived hyperrule-production by some extended consistent replacement as a production of the language consists of three steps:

- 1.) apply some extended consistent replacement  $\psi'_1$  to the sentential form  $\eta$ ;
- 2.) apply another extended consistent replacement  $\Psi'_2$  to a hyperrule  $\gamma$ , and
- 3.) replace in the sentential form a hypernotation  $\langle u \rangle$ , which as a string of metasymbols equals the left hand side of the derived hyperrule  $\gamma(\Psi'_2(\gamma) = \gamma)$ , by the right hand side of that derived hyperrule  $\gamma$ .

Consider the modified derivation in the case of the generation of the word  $\alpha\beta\alpha$ , according to the grammar  $WG_1$ . Take the first hyperrule  $\gamma_1$  as a production of the language (in which case we shall call it hyperrule-production), and apply it to the starting symbol  $\langle s \rangle$ .

Then, the first step of the derivation is:  $\langle s \rangle \Rightarrow \langle aN \rangle \langle bN \rangle \langle aN \rangle \Rightarrow$

In the obtained sentential form replace the metanotation  $N$  by  $nN$  in each of its occurrences, e.g. by applying the extended consistent replacement  $\Psi'_2$  ( $\Psi'_2(N) = nN$ ) to it. Thus, we get

$$\Rightarrow \langle anN \rangle \langle bnN \rangle \langle anN \rangle.$$

Apply hyperrule-production  $\gamma_2$  to hypernotation  $\langle anN \rangle$  of the sentential form in its first occurrence. Replace it by the right hand side of hyperrule-production  $\gamma_2$ . The result will be the sentential form:  $\Rightarrow \alpha \langle aN \rangle \langle bnN \rangle \langle anN \rangle$



Apply the extended consistent replacement  $\Psi_3(\Psi_3(N) = \lambda)$  which is a consistent replacement. We get  $\Rightarrow_{\pi} \alpha \langle a \rangle \langle bn \rangle \langle an \rangle$ .

Replace the notion  $\langle a \rangle$  by  $\lambda$ , applying hyperrule-production  $\gamma_4$ . We get  $\Rightarrow_{\bar{\Gamma}} \alpha \langle bn \rangle \langle an \rangle \Rightarrow_{\bar{\Gamma}}$ . Apply the consistent replacement  $\Psi_5$  to the hyperrule  $\gamma_3$ . Using this production to replace notion  $\langle bn \rangle$  by the right hand side of the derived production, we get  $\Rightarrow_{\bar{\Gamma}} \alpha \beta \langle b \rangle \langle an \rangle \Rightarrow_{\bar{\Gamma}}$ . Replace notion  $\langle b \rangle$  by  $\lambda$  according to production  $\gamma_5$ ; we get  $\Rightarrow_{\bar{\Gamma}} \alpha \beta \langle an \rangle \Rightarrow_{\bar{\Gamma}}$ . Similarly, by replacing notion  $\langle an \rangle$  by  $\alpha \langle a \rangle$ , according to the production obtained from  $\gamma_2$ , and  $\langle a \rangle$  by  $\lambda$  according to the production  $\gamma_4$ , we get  $\Rightarrow_{\pi} \alpha \beta \alpha$ .

A modified derivation step is:

$$\begin{array}{ccc}
 \langle s \rangle \Rightarrow \langle aN \rangle \langle bN \rangle \langle aN \rangle \Rightarrow & \langle anN \rangle \rightarrow \alpha \langle aN \rangle & \\
 \Psi'_1(N) = nN & \Downarrow \text{meta level} & \Downarrow \\
 \langle anN \rangle \langle bnN \rangle \langle anN \rangle & & \langle anN \rangle \rightarrow \alpha \langle aN \rangle \\
 & \text{the application of an hyperrule-rule} & \\
 \Rightarrow \alpha \langle aN \rangle \langle bnN \rangle \langle anN \rangle \Rightarrow & & 
 \end{array}$$

The example was simpler than the real situation: here we did not need the subscription of metanotions, which we are going to define in the following.

The right hand side of the hyperrule-production, derived from a given hyperrule by applying an extended consistent replacement, may contain a certain metanotion which also occurred in the sentential form. Replacing this metanotion by a string obtained from it, according to the metaproducts in each of its occurrences would result only in a sublanguage in the left derivation. In order to avoid this, we supply the metanotions with subscripts the metanotions. Our subscription is similar to the subscription introduced and used in the original definition of the W-grammar [21],[22], but there, only a finite number of subscriptions (namely 10) is allowed.

We shall demonstrate by an example when subscription is necessary. The grammar  $WG_2$  generating the language  $\alpha^j \beta^k$  is

$$\begin{array}{l}
 \text{where } WG_2 = (M_1, \tau_1, \pi_1, \Sigma_1, \Phi_2, \Gamma_2, \langle s \rangle), \\
 \Phi_2 = \{ \langle s \rangle, \langle aa \rangle, \langle bN \rangle, \langle aN \rangle, \langle anN \rangle, \langle bnN \rangle, \langle a \rangle, \langle b \rangle \}, \\
 \text{and} \\
 \Gamma_2 = \{ h_1 : \langle s \rangle \rightarrow \langle aa \rangle \langle bN_1 \rangle, h_2 : \langle aa \rangle \rightarrow \langle aN_2 \rangle, \gamma_2, \gamma_3, \gamma_4, \gamma_5 \}
 \end{array}$$

Apply  $h_1$  to the starting symbol  $\langle s \rangle$  and  $h_2$  to the hypernotation  $\langle aa \rangle$ ; we get the sentential form  $\langle aN \rangle \langle bN \rangle$ . Similarly to the above mentioned method, we will generate words  $\alpha^k \beta^k$  from this sentential form and not the words of the language, generated according to  $WG_2$ . After supplying metanotions with subscripts, the grammar  $WG_2$  will have the hyperrules

$$h_1 : \langle s \rangle \rightarrow \langle aa \rangle \langle bN_1 \rangle, h_2 : \langle aa \rangle \rightarrow \langle aN_2 \rangle \text{ etc.}$$

The modified derivation, with subscribed metanotions, is

$$\langle s \rangle \Rightarrow \langle aa \rangle \langle bN_1 \rangle \Rightarrow \langle aN_2 \rangle \langle bN_1 \rangle \Rightarrow \dots$$



**Definition 3.** A modified W-grammar is an ordered septuple

$$WG' = (M, \tau, \pi, \Sigma, \bar{\Phi}', \bar{\Gamma}', \langle s \rangle)$$

where

$M, \tau, \Sigma$  and  $\langle s \rangle$  are as before, finite sets of metanotions, metaterminals, terminals of the language and the distinguished starting symbol,  $\pi \subset M \times (M \cup \tau)^*$  — a finite set of schemata of context-free metaproducts, and before defining the sets  $\bar{\Phi}'$  and  $\bar{\Gamma}'$ , we introduce the set of subscribed metanotions. Denote by  $I$  the set of the unsigned integer, and by  $M'$  the set of subscribed metanotions (metanotions from set  $M$  and subscript from set  $I$ ).

$$M' = \{A_i : \text{for each } A \in M \text{ and } i \in I\}$$

Furthermore,  $\bar{\Phi}'$  — a finite set of hypernotions,  $\bar{\Phi}' \subset \{\langle u \rangle : u \in (M' \cup \tau)^*\} = \Phi'$ ,  $\bar{\Gamma}' \in \Phi' \times (\bar{\Phi}' \cup \Sigma)^*$  — a finite set of hyperrules.

Denote by  $[u]_k$  the  $k$ -th symbol of the string  $u/u \in (M' \cup \tau)^*$ . Let  $u = a_1 a_2 \dots a_k \dots a_n$ , then  $[u]_k = a_k$ . Denote by  $u'$  the string obtained from  $u$ , by subscribing all metanotions in it

$$[u']_k = \begin{cases} [u]_k & \text{if } [u]_k \in \tau \\ A_i & \text{if } [u]_k = A, A \in M, \text{ for some } i \in I \end{cases}$$

where  $0 < k \leq |u|$ . Denote by  $u'_i$  the set of  $u$ .

The set of metaproducts  $\pi'$  is obtained from the schemata  $\pi$  by subscribing all metanotions in it and adding metaproducts in order to be able to change the subscripts:

$$\pi' = \{A_i \rightarrow A_j : \text{for } i, j \in I, i \neq j \text{ and } A \in M\} \cup \{A_i \rightarrow u' \text{ iff } A \rightarrow u \in \pi \text{ and } i \in I\}$$

For metanotion  $B (B \in M')$  let us define the set

$$L'_B = \{u \in (M' \cup \tau)^* : B \xrightarrow{\pi} u\}$$

**Definition 4.** A semigroup homomorphism  $\Psi' (\Psi' : (M' \cup \tau \cup \Sigma \cup \{<, >, \rightarrow\})^+ \rightarrow (M' \cup \Sigma \cup \{<, >, \rightarrow\})^+)$  is called *extended consistent replacement* iff the following condition is satisfied:

$$\Psi'(x) = \begin{cases} u \alpha L'_x & \text{if } x \in M' \\ X & \text{for all other symbols (as } \tau, \Sigma \text{ and } \{<, >, \rightarrow\}) \end{cases}$$

The hyperrule-productions are obtained from given hyperrules, by applying extended consistent replacements  $\Psi'(\gamma = \Psi'(\bar{\gamma}), \bar{\gamma} \in \bar{\Gamma}')$ .

**Definition 5.** Let  $\eta$  and  $\xi$  be strings over  $(\Phi' \cup \Sigma)$ . We say that  $\xi$  is directly derived from  $\eta$  (by applying one production of the language), iff there are:

- 1.)  $\eta_1, \eta_2$  and  $v (\eta_1, \eta_2 \in (\Phi' \cup \Sigma)^*$  and  $v \in (M' \cup \eta^+)$  for which  $\eta = \eta_1 <v> \eta_2$
- 2.)  $\Psi'_1, \Psi'_2$  extended consistent replacements and a given hyperrule  $<w> \rightarrow \xi \in \bar{\Gamma}'$  for which  $\Psi'_1(v) = \Psi'_2(w)$  and
- 3.)  $\xi = \Psi'_1(\eta_1) \Psi'_2(\xi) \Psi'_1(\eta_2)$ .

Denote this relation by  $\Rightarrow$  and its reflexive and transitive closure by  $\Rightarrow^*$ .

A direct derivation is a direct left derivation iff  $\eta_1 \in \Sigma^*$ ; we denote it by

$\xRightarrow[\Gamma]{1}$ , and its reflexive and transitive closure by  $\xRightarrow[\Gamma]{1}^*$ . The sentential forms are called the items of the following set

$$L''_{WG} = \{\eta \in (\Phi' \cup \Sigma)^* : <s> \xRightarrow[\Gamma]{1} x \eta\}$$

and the language generated by the modified W-grammar WG is the set

$$L(WG) = L'_{WG} \cap \Sigma^* = \{\sigma \in \Sigma^* : <s> \xRightarrow[\Gamma]{1} \sigma\}$$

**Definition 6.** An extended consistent replacement  $\Psi'$  is restricted (non restricted) relative to string  $\xi$  by metanotion C, iff there are (are not) two different metanotions A, B ( $A, B \in M'$  and  $A \neq B$ ) contained in string  $\xi$  ( $|\xi|_A \neq 0$ ), the images of which under  $\Psi'$  (would) contain the same metanotions with equal subscripts. I.e. there is (is not) some metanotion C, for which the systems of inequality  $|\Psi'(A)|_C \neq 0$  and  $|\Psi'(B)|_C \neq 0$  are satisfied.

**Definition 7.** Two extended consistent replacements  $\Psi'_1$  and  $\Psi'_2$  are called restricted, relative to the strings  $\xi_1$  and  $\xi_2$  by metanotion C, iff the following conditions are satisfied:

- 1.) If  $\Psi'_1$  is restricted, relative to  $\xi_1$  by some C, then  $\Psi'_2$  is nonrestricted, relative to  $\xi_2$  by C, and vice versa.
- 2.) The strings  $\xi_1$  and  $\xi_2$  contain such metanotions A and B, the images of which under  $\Psi'_1$  and  $\Psi'_2$  ( $\Psi'_1(A), \Psi'_2(B)$ ) contain the same metanotion C; i.e.

$$|\Psi'_1(A)|_C \neq 0 \text{ and } |\Psi'_2(B)|_C \neq 0$$



Let us assume without loss of generality that the extended consistent replacements  $\Psi_1$  and  $\Psi'_2$  in any direct derivation (of the sentential form) are restricted, relative to  $\eta$  and  $\xi$  by certain metanotion  $C$ , iff the replaced hypernotation contains  $C$ ,

$$|\Psi'_1(v)|_C \neq 0.$$

Let the pair of extended consistent replacements  $\Psi'_{11}$  and  $\Psi'_{12}$ , also satisfy the conditions of direct derivation:

$$\eta = \eta_1 \langle v \rangle \eta_2, \Psi'_{11}(v) = \Psi'_{12}(w) \text{ and } \xi = \Psi'_{11}(\eta) \Psi'_{12}(\xi) \Psi'_{11}(\eta_2)$$

**Definition 8.** The pair of extended consistent replacements  $\Psi'_{11}$  and  $\Psi'_{12}$ , relative to the string  $v$  and  $w$ , iff there is an extended consistent replacement  $\Psi'$ , for which

$$\Psi'(\Psi'_{11}(v)) = \Psi'_{11}(u) \text{ and } \Psi'(\Psi'_{12}(w)) = \Psi'_{12}(w)$$

**Definition 9.** Two pairs of extended consistent replacements  $\Psi'_1, \Psi'_2$  and  $\Psi'_{11}, \Psi'_{12}$  are called *independent*, iff none of them is a pair of subreplacements of the other one relative to  $v$  and  $w$ .

For example, when the hyperrule-production was applied to the sentential form  $\langle aN \rangle \langle bN \rangle \langle aN \rangle$ , i.e. when the extended consistent replacements  $\Psi'_1$  and  $\Psi'_2$  were used for which  $\Psi'_1(N) = nN$  and  $\Psi'_2(N) = N$ , then we could take another pair of extended consistent replacements  $\Psi'_{11}, \Psi'_{12}$  defined by the equations  $\Psi'_{11}(N) = nnN$ . But this pair depends on the pair  $\Psi'_1, \Psi'_2$  as there is an extended consistent replacement for which  $\Psi'(N) = nN$  and so

$$\Psi'(\Psi'_1(v)) = \Psi'(\Psi'_1(aN)) = \Psi'(anN) = annN \Psi'_{11}(aN)$$

and

$$\Psi'(\Psi'_2(w)) = \Psi'(\Psi'_2(anN)) = \Psi'(anN) annN = \Psi'_{12}(anN)$$

### 2.3. Equivalency of two definitions of the derivation by the W-grammar

**Theorem 1.** The languages generated by  $WG$  and  $WG'$  are the same, and the hyperrules, from which the productions of the language and metaderived hyperrule-productions are derived and applied in the derivations of a word of the language, are the same.

**Proof:** By our assumption, a metanotion will be contained in the sentential form at least twice with the same subscript, iff it was contained at least twice with the same subscript by the right hand side of a hyperrule-production, applied in some step of the derivation; or it was contained in a metastring, which replaced a metanotion (contained at least twice by the sentential form with the same subscript), when an extended consistent replacement was applied



according to the modified derivation by W-grammar, because only those metanotions have to be replaced by the same metastring, which were contained in the right hand side of some applied hyperrule-production and which were equal and had the same subscript, i.e. which are replaced by the same metastring to the original definition of W-grammar.

On the other hand, if some metanotion was contained at least twice in the right hand side of a hyperrule of WG, then it will be contained in the sentential form at least twice with the same subscript, and metanotions contained in the string which replaced the metanotion, will have the same subscript standing in the same place of the string, and so they have to be replaced by the same metastrings.

The modified definition of the derivation by the W-grammar is more general, as each derivation according to the original definition, is a derivation according to the modified definition of the derivation with the addition of subscripts.

$$L_X \subset L_X \subseteq L_{x_i} \quad \text{for some subscript } i \in I.$$

The reverse assertion, i.e. each derivation of a terminal string of the language according to the modified derivation, being in correspondence with the derivation of the same terminal string in the original sense, is proved by induction on the steps of derivation. The first step is trivial, as each hyperrule for the starting symbol of the language is in correspondence with a production derived from it.

Assume that this is true until the  $t$ -th step, after which we have the sentential form  $\eta$ . Any string derived from  $\eta$  by consistent replacement, can be derived from the starting symbol, according to the original definition of the derivation; and any derivation, according to the original definition of the derivation by W-grammar is in correspondence with a modified definition. We shall prove that this will be true for the string directly derived from the sentential form  $\eta$  by application of a hyperrule-production, metaderived from the given hyperrule  $\langle w \rangle \rightarrow \zeta \epsilon \Gamma'$ .

Applying a hyperrule-production, metaderived from  $\langle w \rangle \rightarrow \xi$  means that there are  $\eta_1, \eta_2 \in (\Phi' \cup \Sigma)^*$ ,  $v \in (M' \cup \tau)^*$  and the pair of the extended consistent replacements  $\Psi'_1, \Psi'_2$  for which the following equations are satisfied:

$$\eta = \eta_1 \langle v \rangle \eta_2, \quad \Psi'_1(v) = \Psi'_2(w), \quad \xi = \Psi'_1(\eta_1) \Psi'_2(\xi) \Psi'_1(\eta_2)$$

Assume that some string  $\mu$  of the notions can be derived from the sentential form  $\eta$  by a consistent replacement  $\Psi'(\mu = \Psi'(\eta))$ , which string cannot be derived from the starting symbol according to WG. The applicability of the hyperrule-production  $\langle \Psi'_2(w) \rangle \rightarrow \Psi'_2(\xi)$  to a sentential form, means that all hyperrule-productions metaderived from  $\langle \Psi'_2(w) \rangle \rightarrow \Psi'_2(\xi)$  are also applicable to  $\eta$ ; i.e. hyperrule-production

$$\langle \Psi'(\Psi'_2(w)) \rangle \rightarrow \Psi'(\Psi'_2(\xi)) \text{ is applicable, too.}$$



A sentential form containing only notions, is obtained by the sequential application of the extended consistent replacements  $\Psi', \Psi'_1$ . This sentential form  $\mu$  can be derived from the starting symbol, according to WG, by the assumption of induction. For that string  $\mu$ , let us apply production  $\langle \Psi'(\Psi'_2(w)) \rangle \rightarrow \Psi'(\Psi'_2(\xi))$ , and we get  $\mu$ . It contradicts our last assumption, that there is a sentential form of modified derivation, which is not a sentential form according to the original definition.

### 3. Extended Recursive Descent and Its Use

Let us describe the extension of the well-known parsing method called recursive descent and demonstrate its work on the example grammar  $WG_1$  introduced in Section 2.1.

In Section 3.2. the conditions of well-formed W-grammars will be described. Finally we modify W-grammar L3, describing the language ASPLE, given in [4].

#### 3.1. Extensions of recursive descent

Consider a CF grammar  $G$ , productions of which are given in the BNF form.

A parser for  $G$ , produced by the method of recursive descent, is a program made up of a family of procedures, calling each other recursively; the program itself is one of these procedures. Each procedure attempts to find an occurrence of a particular syntactic type in the input, a substring derived from a nonterminal to which the procedure corresponds. The called procedure returns with the value "true" or "false", depending on whether it has been successful or not.

In the method of extended recursive descent the hypernotations defined by hyperrules having BNF-like form, are made up (as in the case of CF grammars) of procedures, calling each other recursively. They have parameters which actually are expressions of so-called grammar-type variables and constants.

A grammar-type constant is such a constant which corresponds to a metaterminal of the W-grammar (in a one-to-one correspondence).

Grammar-type variables are variables the set of values of which is defined by the metanotions of the W-grammar. We say a variable is related to a metanotion  $A$  (the type of the variable is the grammar-type  $A$ ), iff the set of values of the variable is a set of expressions of grammar-type variables and constants, whose related strings (the symbols of which correspond or relate to the operands of expressions in order of their appearance) can be derived, by applying metarules, from  $A$ .

An expression is called a *grammatical expression* if its operands are grammar-type variables and constants joined to the operation concatenation. The formal and actual parameters of procedures are grammatical expressions. Two grammatical expressions are transformable into a common expression, iff there are such assignations of variables of expressions, after the execution of



which they turn into identical expressions.

*Parameter passing* is allowed, iff actual and formal parameters which are grammatical expressions, can be transformed into a common expression. Parameter passing means this transformation i.e. execution of assignments resulting in identical expressions.

In the case of recursive descent, the corresponding between nonterminals and procedures is one-to-one. In our case, however, a hypernotation corresponds to every procedure which can be called with the actual parameter (being a grammatical expression), related to the hypernotation. To call a procedure in the extended recursive descent, means to call all procedures and check the parameter passes. It is sufficient to call only those procedures, the formal parameters of which are transformable with the actual grammatical expression into a common expression.

Consider the analyser constructed by  $WG_1$  in the above described way:

$\gamma_1 : \langle s \rangle \rightarrow \langle aN_1 \rangle \langle bN_1 \rangle \langle aN_1 \rangle$

P1: declare  $N_{i_N}$ ;

form the tree of expression  $aN_{i_N}$  and call procedures P2, P4 recursively until the first true result and if the result of call is false (none of them was true) then exit;

form the tree of  $bN_{i_N}$  and if not (P3 or P5) then exit;

form the tree  $aN_{i_N}$  and if not (P2 or P4) then exit;

$i_N = i_N + 1$  and return with result true;  
exit: return with the value false.

$\gamma_2 : \langle aN_2 \rangle \rightarrow \alpha \langle aN_2 \rangle$

P2: declare  $N_{i_N}$ ;

make the tree of the formal parameter-expression  $aN_{i_N}$

and pass grammar-type parameter (call ANAL);

if the next character  $s_j \neq a$  then exit  
else  $j = j + 1$ ; form the tree  $aN_{i_N}$  and if not (P2 or P4) then exit;

$i_N = i_N + 1$  and return with true;

exit: return with false.



$\gamma_3 : \langle \text{bnN}_3 \rangle \rightarrow \beta \langle \text{bN}_3 \rangle$

P3: declare  $N_{i_N}$ ;

form  $\text{bnN}_{i_N}$  and if not ANAL then exit;

if  $s_j \neq \beta$  then exit else  $j = j + 1$ ;

if not (P3 or P5)  $\text{bN}_{i_N}$  then exit;

$i_N = i_N + 1$  and return with true;

exit: return with false.

$\gamma_4 : \langle a \rangle \rightarrow \lambda$

P4: if not ANAL(a) then exit;

return with true;

exit: return with false.

$\gamma_5 : \langle b \rangle \rightarrow \lambda$

P5: if ANAL(b) then return with true

else return with false.

Consider, how the word  $\alpha\beta\alpha$  generated by  $WG_1$  can be parsed by this analyser. Mark the current character of  $\alpha\beta\alpha$  by dot:  $\alpha\beta\alpha$ . The first called procedure is P1, corresponding to  $\gamma_1$ , defining the starting symbol of  $WG_1$

(1 — 1 where the first number is the level of the procedure calls; the second number separated from the first one by a hyphen, is the sequence of subscripts  $n$  of  $P_n$  of the called procedures)

Call P1 with the input string  $\alpha\beta\alpha$ ,

1. The first step is the declaration of the grammar-type variable  $N_1$  (the subscript marks the place, where the metanotation appears in the derivation) related to metanotation  $N$ .

2. Next, call procedures recursively with the actual parameter-expression — hypernotation  $aN_1$  (grammar-type variables and constants will be marked by the same letter as the metasymbols related to them). It is sufficient to call only those ones applicable in principle (when we speak in the following of applicable hyperrules or procedures, related to the same hyperrules, they are to be understood as hyperrules applicable in principle, too).

P2, P4.(2 — 12) Call P2( $aN_1$ ) with  $\alpha\beta\alpha$ .

1. In the procedure which is the implementation of hyperrule  $\gamma_2$ , the variable to be declared is  $N_2$ .

2. The formal parameter of procedure P2 is expression  $aN_2$ . The assignation routine named ANAL called after the declaration of  $N_2$ , with parameters  $aN_1$  and  $aN_2$  will find that

- metaterminals 'a' are equal;
- string-expression  $nN_2$  can be derived from metanotion  $N_1$ ;
- parameter passing is completed; the value of variable  $N_1$  after assignation is shown in Fig. 1(a).

3. The terminal of the language  $\alpha$  is compared with the current character of the input string. As they are equal  $\alpha$  is read.

4. The next expression is  $aN_2$  and applicable procedures are P2 and P4. (3-122) call P2 with  $\alpha.\beta\alpha$ .

1. Declare  $N_3$  (The used metaproduction is  $N_2 \rightarrow N_3$  applied to change subscription).
2. Call ANAL with  $aN_2$  and  $aN_3$  which will assign value  $nN_3$  to  $N_2$ . The value  $N_1$  is shown in Fig. 1(b).

3. Compare  $\alpha$  and current symbol  $\beta$ . They are not equal and as there is no next alternative, exit from the procedure; unmake assignation  $nN_1$  to  $N_2$  and return with value "false".

The next procedure to be tried is:

P4. (3 - 124) Call P4 with  $\alpha.\beta\alpha$ .

1. Nothing to declare (since the corresponding hyperrule does not contain metanotions).
2. Call ANAL ( $aN_2, a$ ) which will assign value  $\lambda$  to  $N_2$ .
3. Return with the value "true" from  $P_4$ .

As in  $P_2$  the call with expression  $\langle aN_2 \rangle$  was the last one,  $P_2$  will return with the value "true".

The variable  $N_1$  will have value  $n$ , the tree structure of which is demonstrated in Fig. 1.(C).



The next hypernotation of  $\gamma_1$  is  $\langle bN_1 \rangle$ , the implementation of which is expression  $\langle bN_1 \rangle$ , in which the operand  $N_1$  obtained value  $n$  and so the actual parameter is  $\langle bn \rangle$ . Applicable hyperrules are:

$\gamma_3, \gamma_5$ . 2-1243 Call  $\gamma_3$  with  $\alpha.\beta\alpha -|$ .

1. Declare  $N_4$ .
2. Call ANAL ( $bn, bnN_4$ ), which will assign value  $\lambda$  to  $N_4$ .
3. Compare  $\beta$  with the current character of the input string; as they are equal, the current character must be skipped.
4. The value of expression  $\langle bN_4 \rangle$  is  $\langle b \rangle$  and applicable hyperrules are:

$\gamma_3, \gamma_5$ . 2 - 12433 Call  $\gamma_3$  with  $\alpha.\beta\alpha -+$ .

1. Declare  $N_5$ .
2. Call ANAL ( $b, bnN_5$ ).

- the first metaterminal symbols are equal
- the end of the expression and the metaterminal  $n$  are not equal and there is no assignation making them equal. It returns with the value "false".

Procedure  $\gamma_3$  cannot be called with the actual parameter  $\langle b \rangle$ .

The next applicable hyperrule is:

$\gamma_5$ . (3-12435) Call  $\gamma_5$  with  $\alpha\beta.\alpha -|$ .

1. Nothing to declare.
2. Call ANAL ( $b, b$ ).
3. It returns with the value "true".

The next expression of  $P_1$  is expression  $\langle aN_1 \rangle$  which has actually the value  $\langle an \rangle$ . The parse of the string (namely  $\alpha$ ) derived from notion  $\langle an \rangle$  is similar to the parse of  $\beta$  from  $\langle bn \rangle$ .

The parsing ends: when the end symbol  $-|$  is reached.

**Algorithm:** An algorithm, which can be used in ANAL.

Input: the expressions  $v$  and  $w$ , the actual items of which are marked with point symbols,  $\underline{v} = .v -|$  and  $\underline{w} = .w -|$ .

*method:* Step 1. Skip equal metaterminals. There are metastrings ,



$v_1, v_2, w_1, w_2 \in (M' \cup \tau)^*$  and  $c \in \tau$ , for which  $\underline{v}_i = v_1 \cdot c v_2 \dashv$  and  $\underline{w}_j = w_1 \cdot c w_2 \dashv$  where  $1 \leq i \leq |v| + 1$  and  $1 \leq j \leq |w| + 1$ . Skip metaterminal  $c$  in both strings  $\underline{v}_i, \underline{w}_j$ , the result strings are  $\underline{v}_{i+1} = v_1 c \cdot v_2 \dashv$  and  $\underline{w}_{j+1} = w_1 c \cdot w_2 \dashv$ .

*Step 2. Metaderivation.* There are  $v_1, v_2, v_3, w_1, w_2 \in (M' \cup \tau)^*$ ,  $B \in M'$  for which  $\underline{v}_i = v_1 \cdot v_2 v_3 \dashv$ ,  $\underline{w}_j = w_1 \cdot B w_2 \dashv$  and  $B \xRightarrow{\pi} v_2$ , then  $\underline{v}_{i+|v_2|} = v_1 v_2 \cdot v_3 \dashv$  and  $\underline{w}_{j+1} = w_1 B \cdot w_2 \dashv$ , assign the value  $v_2$  to  $B$ , and save the reference to the assignment.

*Step 3. Metaderivation.* There are  $v_1, v_2, w_1, w_2, w_3 \in (M \cup \tau)^*$ ,  $A \in M'$ , for which  $\underline{v}_i = v_1 \cdot A v_2 \dashv$ ,  $\underline{w}_j = w_1 \cdot w_2 w_3 \dashv$  and  $A \xRightarrow{\pi} w_2$ , then  $\underline{v}_{i+1} = v_1 A \cdot v_2 \dashv$  and  $\underline{w}_{j+|w_2|} = w_1 w_2 \cdot w_3 \dashv$ , assign the value  $w_2$  to  $A$  and save the reference to the assignment.

*Step 4.* Stop with value "true", when  $v_{v+1} = v \cdot \dashv$  and  $w_{w+1} = w \cdot \dashv$

*Step 5.* Stop with value "false".

### 3.2. Well-formed (considering algorithm ANAL) W-grammar

The problem of finding a hyperrule from which such a production of the language can be derived which is applicable to a notion derived from a hypernotation, is undecidable in general, as it is equivalent to the problem of finding the intersection of languages generated by two context-free grammars, which is in turn undecidable. We will deal only with cases where this problem is decidable, because in the definitions used in practice this problem must be decidable.

We are now going to study *three groups* of intuitively necessary conditions on W-grammars which must be satisfied so that the parser made by W-grammar should work and stop with the value "true" only for input strings derived by the W-grammar. The proof of the conditions, as stated in the introduction will be detailed in a forthcoming paper.

#### Group 1. Restrictions if the infinite loops.

**Definition 10.** A hypernotation  $\langle v \rangle$  is called *n times cyclic*, iff there is a string  $\xi/\xi \in (\Sigma \cup \Phi')^*$ , in the left derivation (from  $\langle v \rangle$ ) of which  $n$  hyperrule-rules derived from some given hyperrule  $\langle w \rangle \rightarrow \xi \in \Gamma'$  were applied to the left symbol-hypernotation of the derived strings; there are  $\eta_1, \dots, \eta_n, \mu_1, \dots, \mu_n$ , and  $u_1, \dots, u_n$  for which

$$\langle v \rangle \xRightarrow{\ell} * \langle u_1 \rangle \eta_1 \xRightarrow{\ell} \mu_1 \eta_1 \xRightarrow{\ell} * \langle u_2 \rangle \eta_2 \xRightarrow{\ell} \mu_2 \eta_2 \xRightarrow{\ell} * \dots \xRightarrow{\ell} * \langle u_n \rangle \eta_n \xRightarrow{\ell} \mu_n \eta_n = \xi$$

and the applied hyperrule-rules were derived from the given hyperrule  $\langle w \rangle \rightarrow \xi$  (i.e. there are  $\Psi'_{i1}, \Psi'_{i2}$ , for which  $\Psi'_{i1}(u_i) = \Psi'_{i2}(w), \mu_i = \Psi_{i2}(\xi), \bar{\eta}_i = \Psi'_{i1}(\eta_i), 1 \leq i \leq n$ ).



Hypernotation  $\langle v \rangle$  is not  $n + 1$  times cyclic.

**Condition 1.** For every string there is a number  $k$  which is the maximum value of the degree of cycled hypornotions in the left derivation of these strings.

**Group 2.** The algorithm ANAL will find the necessary assignments, if every direct left derivation of any  $\xi$  from any  $\eta$  satisfies the following conditions 2 - 6.

Recall the definition of the direct left derivation  $\xi$  from  $\eta$ :

1. There are  $\eta_1 \in \Sigma^*$ ,  $\eta_2 \in (\Sigma \cup \Phi')^*$  and  $v \in (M' \cup \tau)^*$  for which  $\eta = \eta_1 \langle v \rangle \eta_2$
2. There are  $\Psi'_1, \Psi'_2$  and  $\langle w \rangle \rightarrow \xi \in \bar{\Gamma}'$  for which  $\Psi'_1(v) = \Psi'_2(w)$
3.  $\xi = \Psi'_1(\eta_1) \Psi'_2(\xi) \Psi'_1(\eta_2)$ . All conditions will relate to this application.

**Condition 2.** There is a metastring  $u \in (M' \cup \tau)^*$ , for which  $L'_{v \cap} L'_w = L'_u$

**Condition 3.** There is only one independent pair  $\Psi'_1, \Psi'_2$  for the application of a hyper-production derived from  $\langle w \rangle \rightarrow \xi$ .

**Condition 4.** There is only one grouping  $v$  and  $w$ :  $v = X_1 \dots X_n$ ,  $w = Y_1 \dots Y_n$  ( $X_i, Y_i \in (M' \cup \tau)$  for  $1 \leq i \leq n$ ), for which

1. if  $X_i \in \tau$  and  $Y_i \in \tau$  ( $1 \leq i \leq n$ ), then  $X_i = Y_i$ , or
2.  $X_i \in (M' \cup \tau)^*$ ,  $Y_i \in M'$  and then  $\Psi'_2(Y_i) = X_i$  (or  $\Psi'_2(Y_i) = \Psi'_1(X_i)$ ) but then  $\Psi'_1$  for all metanotions from substrings  $X_i$  changes only the subscripts).
3.  $X_i \in M'$ ,  $Y_i \in (M' \cup \tau)^*$ , and then  $\Psi'_1(X_i) = Y_i$  (or  $= \Psi'_2(Y_i)$ ) but then  $\Psi'_2$  changes only subscripts for all metanotions from  $Y_i$ .

**Condition 5.** There is no such subscript  $i$  ( $1 \leq i \leq n$ ), for which, if  $X_i = \Psi'_2(Y_i)$  (or  $\Psi'_1(X_i) = Y_i$ ) there are  $v_1, v_2 \in (M' \cup \tau)^*$  and  $\Psi'$ , for which  $X_{i+1} = v_1 v_2$  (or  $Y_{i+1} = v_1 v_2$ ) and  $X_i v_1 = \Psi'(Y_i)$  (or  $\Psi'(X_i) = Y_i v_1$ ).

**Condition 6.** There is no  $i$  ( $1 \leq i \leq n + 1$ ), for which  $\Psi'_1(X_i) = Y_i$  and  $\lambda \in L_{X_i}$ ,  $\lambda \in L_{Y_i}$  at the same time.

**Group 3.** The conditions which must be satisfied when the no-backup method works. Let us define the set:

$$L_{v, A, \xi}^{WG} = \{u : \langle v \rangle = \xi \xRightarrow[0]{WG} \xi \xRightarrow[1]{WG} \dots \xRightarrow[n]{WG} \xi_n = \xi, \xi \in \Sigma^*, \Psi'_{n+1}(\dots(\Psi'_{11}(A))\dots) = u, A \in M', |u|_A \neq 0\}$$

**Condition 7.** The set  $L_{v,A,\xi}^{WG}$  contains only one item for any derivation, i.e.

$$|L_{v,A,\rho}^{WG}| = 1$$

Let us define the sets

$$FIRST_{WG}(v) = \{a \in \Sigma : \langle v \rangle \Rightarrow_{WG}^* a\xi, \xi \in (\Phi' \cup \Sigma)^*\}$$

$$FOLLOW_{WG}(v) = \{a \in \Sigma : \langle s \rangle \Rightarrow_{WG}^* \eta \langle v \rangle a\xi, \eta, \xi \in (\Phi' \cup \Sigma')^*\}$$

Let the hyperrules of W-grammar have one of the following forms (see [5] for the idea how it can be done):

1.  $\langle v \rangle \rightarrow \langle w \rangle \mid \langle u \rangle$
2.  $\langle v \rangle \rightarrow \langle w \rangle \langle u \rangle$
3.  $\langle v \rangle \rightarrow \langle w \rangle$
4.  $\langle v \rangle \rightarrow a$
5.  $\langle v \rangle \rightarrow \lambda$

When the no-backup method works (described in [12]) then the following conditions have to be satisfied:

**Condition 8.** For every hyperrule of type 1,  $FIRST(w) \cap FIRST(u) = \emptyset$

**Condition 9.** For every hyperrule of type 1, where  $u$  can produce the empty string,  $FIRST(w) \cap FOLLOW(v) = \emptyset$

**Condition 10.** For every hyperrule of type 1,  $\langle w \rangle$  is not such a hypernotation from which the empty string can be derived.

### 3.4. Example: Modified ASPLE.

First, we recall the definition of ASPLE, as described and defined by definition 3 in [4], in which the syntax and the context-conditions were defined by W-grammar  $L3$ .



### Metaproductions

- (L3.A) ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o;  
p; q; r; s; t; u; v; w; x; y; z.
- (L3.B) NOTION :: ALPHA; NOTION ALPHA.
- (L3.C) EMPTY :: .
- (L3.D) NOTETY :: NOTION; EMPTY.
- (L3.E) LETTER :: letter ALPHA.
- (L3.F) TAG :: LETTER; TAG LETTER.
- (L3.G) DEF :: and TAG has MODE.
- (L3.H) DEFS :: DEF; DEFS DEF.
- (L3.I) DEFSETY :: DEFS; EMPTY.
- (L3.J) TABLE :: DEFS.
- (L3.K) MODE :: REFSETY INTBOOL.
- (L3.L) DELIMITER :: go on; comma; plus; times.
- (L3.M) REFS :: ref; REFS ref.
- (L3.N) REFSETY :: REFS; EMPTY.
- (L3.O) INTBOOL :: int; bool.
- (L3.P) ALPHABET :: abcdefghijklmnopqrstuvwxyz.

### Hyper-rules

- (L3.1) true : EMPTY.
- (L3.2) where NOTETY is NOTETY : true.
- (L3.3) where NOTETY1 NOTION NOTETY2 contains NOTION :  
true.
- (L3.4) where NOTETY1 ALPHA1 differs from NOTETY2 ALPHA2 :  
where NOTETY1 differs from NOTETY2;  
where ALPHA1 precedes ALPHA2 in ALPHABET;  
where ALPHA2 precedes ALPHA1 in ALPHABET.
- (L3.5) where ALPHA1 precedes ALPHA2 in NOTETY1 ALPHA1  
NOTETY2 ALPHA2 NOTETY 3 : true.
- (L3.6) where NOTION differs from EMPTY : true.
- (L3.7) where EMPTY differs from NOTION : true.
- (L3.8) NOTION sequence :  
NOTION;  
NOTION sequence, NOTION.
- (L3.9) NOTION list separated by DELIMITER :  
NOTION;  
NOTION list separated by DELIMITER,  
DELIMITER symbol, NOTION.
- (L3.10) NOTION pack :  
left paren symbol,  
NOTION,  
right paren symbol.
- (L3.11) program :  
begin symbol,  
declare of TABLE,  
TABLE statement train,  
end symbol,  
TABLE restriction.
- (L3.12) declare of DEFS DEFSETY :  
MODE declarer,  
ref MODE definitions of DEFS,  
go on symbol, declare of DEFSETY;  
where DEFSETY is EMPTY, MODE declarer,  
ref MODE definitions of DEFS, go on symbol.

- (L3.13) ref MODE declarer : ref symbol, MODE declarer.
- (L3.14) INTBOOL declarer : INTBOOL symbol.
- (L3.15) MODE definitions of and TAG has MODE DEFSETY :  
     TAG identifier, comma symbol,  
     MODE definitions of DEFSETY;  
     where DEFSETY is EMPTY,  
     TAG identifier.
- (L3.16) DEFSETY and TAG has MODE restriction:  
     where TAG is not in DEFSETY,  
     DEFSETY restriction;  
     where DEFSETY is EMPTY.
- (L3.17) where TAG1 is not in and TAG2 has MODE DEFSETY :  
     where TAG1 differs from TAG2,  
     where TAG1 is not in DEFSETY;  
     where DEFSETY is EMPTY,  
     where TAG1 differs from TAG2.
- (L3.18) TABLE statement train :  
     TABLE statement list separated by go on.
- (L3.19) TABLE statement :  
     TABLE assignment;  
     TABLE conditional;  
     TABLE loop;  
     TABLE transput.
- (L3.20) TABLE assignment :  
     TABLE ref MODE identifier,  
     becomes symbol,  
     TABLE MODE value.
- (L3.21) TABLE conditional :  
     if symbol,  
     TABLE bool value,  
     then symbol,  
     TABLE statement train,  
     TABLE elsend.
- (L3.22) TABLE elsend :  
     fi symbol;  
     else symbol,  
     TABLE statement train,  
     fi symbol.
- (L3.23) TABLE loop :  
     while symbol,  
     TABLE bool value,  
     do symbol,  
     TABLE statement train,  
     od symbol.
- (L3.24) TABLE transput:  
     in symbol,  
     strong TABLE ref INTBOOL identifier;  
     out symbol,  
     TABLE INTBOOL value.
- (L3.25) TABLE ref MODE value :  
     strong TABLE ref MODE identifier.
- (L3.26) TABLE INTBOOL value :  
     TABLE INTBOOL factor list separated by plus.
- (L3.27) TABLE INTBOOL factor :  
     TABLE INTBOOL primary list separated by times.
- (L3.28) TABLE INTBOOL primary :  
     INTBOOL denotation;  
     strong TABLE INTBOOL identifier;  
     TABLE INTBOOL value pack;  
     where INTBOOL is bool ,  
     TABLE compare pack.
- (L3.29) TABLE compare :  
     TABLE int value,  
     relation,  
     TABLE int value.



- (L3.30) relation : equals symbol; not equals symbol.
- (L3.31) int denotation : digit sequence.
- (L3.32) bool denotation : true symbol; false symbol.
- (L3.33) digit : zero symbol; one symbol; two symbol;  
three symbol; four symbol; five symbol;  
six symbol; seven symbol; eight symbol;  
nine symbol.
- (L3.34) strong TABLE MODE identifier :  
strong TABLE ref MODE identifier;  
TABLE MODE identifier.
- (L3.35) TABLE MODE identifier :  
TAG identifier,  
where TABLE contains and TAG has MODE.
- (L3.36) LETTER TAG identifier : LETTER symbol, TAG identifier.
- (L3.37) LETTER identifier : LETTER symbol.

We will not deal with the modification of metarules in detail, because it depends on the parser used for the metasystem, but it will differ from the usual parsers made for CF grammars, in that it recognizes defining items for grammar-type variables as well. We will give only metarules that cause the recognition of certain combinations of grammar-type variables.

The hyperrule LM.1 is the unchanged hyperrule L3.1.

The use of the hyperrule L3.2 does not satisfy Condition 5, the metaterminal  $i$  is contained in the language derived from NOTION ( $i \in L_{\text{NOTION}}$ ), the derivation (Step 2) will not stop when the current character is  $i$ . This problem can be avoided by separating metaterminal  $i$  from NOTION by brackets. The modified hyperrule is:

LM.2 where (NOTION) is (NOTION): true. , and the hyperrules 12, 16, 17, and 28 must be modified accordingly.

The use of L3.3 does not satisfy the same Condition (5), but its modification will differ from the previous one, because the end of the metastring derived from NOTETY1 depends on the value of the NOTION dynamically. In the modified grammar the hyperrule L3.3 is replaced by the following two hyperrules:

LM.3.1 where (NOTION) is contained in NOTION NOTETY: true.

LM.3.1 where (NOTION) is contained in ALPHA NOTION1: where (NOTION) is contained in NOTION1., and the hyperrules containing hypernotations which can be derived from the hypernotation <where NOTION contains NOTION1> are also modified accordingly.

The hyperrule L3.4 must be modified for the same reason as L3.2 and L3.3, in the following way:

LM.4 where (ALPHA1 NOTETY1) differs from ALPHA2 NOTETY2:  
where (NOTETY1) differs from NOTETY2;  
where ALPHA1 precedes ALPHA2 in ALPHABET;  
where ALPHA2 precedes ALPHA1 in ALPHABET.



- LM.5    where ALPHA1 precedes ALPHA2 in ALPHA NOTION:  
         where (ALPHA1) is (ALPHA), where (ALPHA2) is contained  
         in NOTION:  
         where ALPHA1 precedes ALPHA2 in NOTION.
- LM.6    where NOTION differs from EMPTY: true.
- LM.7    where EMPTY differs from NOTION: true.
- LM.8    NOTION sequence: NOTION, NOTION sequence.
- LM.9    NOTION list separated by DELIMITER: NOTION;  
         NOTION, DELIMITER symbol, NOTION list separated by DELIMITER.
- LM.10   packed NOTION: left paren symbol, NOTION, right paren symbol.  
         The hyperrules LM.11-LM.14 are the unchanged hyperrules L3.11-L3.14,  
         but the metaproduction:
- DEFS → DEFS DEFSETY
- must be added to Condition 2.
- LM.15   MODE definitions of and TAG has MODE DEFSETY: TAG identifier,  
         comma symbol, MODE definitions of DEFSETY;  
         where DEFSETY is EMPTY, TAG identifier.
- LM.16   and TAG has MODE DEFSETY restriction: where DEFSETY is  
         EMPTY; where TAG is not in DEFSETY, DEFSETY, restriction.
- LM.17   where TAG1 is not in TAG2 has MODE DEFSETY:  
         where TAG1 differs from TAG2, where TAG1 is not in DEFSETY;  
         where DEFSETY is EMPTY, where TAG1 differs from TAG2.
- LM.18   TABLE statement train: TABLE statement list separated by go on.  
         The hyperrules LM.19-LM.25 and the hyperrules L3.19-L3.25 are the  
         same.
- LM.26   TABLE INTBOOL value: TABLE INTBOOL factor list separated by plus.
- LM.27   TABLE INTBOOL factor: TABLE INTBOOL primary list separated by times.
- LM.28   TABLE INTBOOL primary: INTBOOL denotation; strong TABLE INTBOOL  
         identifier; packed TABLE INTBOOL is bool, packed TABLE compare.  
         The hyperrules L3.29-L3.34 need not be modified.
- LM.35   TABLE MODE identifier: TAG identifier, where and TAG has MODE are  
         contained in TABLE.



The hyperrules L3.36 and L3.37 can go to the new set without changes, and the metaproductions

TAG → LETTER TAG

must be added.

The modification of grammar L3 is not significant from the view-point of definition, but it is very important for parsing.

### R e f e r e n c e s

- [1] Baker, J.L.: Grammars with Structured Vocabulary: A Model for the ALGOL 68 Definition. *Information and Control*, 20 (1972), 351-395.
- [2] Brattchikov, I.I.: Syntax of Programming Languages. In Russian. Nauka, 1975.
- [3] Cejtin, G.S.: ALGOL 68. Methods of Implementation. In Russian. State University Leningrad (LGU), 1976.
- [4] Cleveland, J.C., Uzgalis, R.C.: Grammar for Programming Languages. Elsevier North-Holland. 1977.
- [5] Deussen, P.A.: Decidability Criterion for van Wijngaarden Grammars. *Acta Informatica* 5 (1975), 353-375.
- [6] Deussen, P.A., Mehlhorn, K.: Van Wijngaarden Grammars and Space Complexity Class EXPACE. *Acta Informatica* 8 (1978), 193-199.
- [7] Farkas, E.: Comparison of some Methods for the Definition of Static Semantics. *Computational Linguistics and Computer Languages XIII*. (1970), 7-18.
- [8] Gerevich, L.: A Parser for van Wijngaarden Grammars. In Russian. Proceedings of Computer and Automation Institute, Hungarian Academy of Sciences MTA SzTAKI, *Közlemények* 21/1978, 7-20.
- [9] Gerevich, L.: An Analysis-Oriented-Definition of van Wijngaarden Grammars. In Hungarian, Papers of Research Institute for Applied Computer Sciences, SzTAKI *Tanulmányok* 4/1979, 31-39.
- [10] Greibach, S.A.: Some Restrictions on W-Grammars. *ACM Proceedings of the 6th Symposium on the Theory of Computing*, Seattle (1974).
- [11] Hesse, W.: A Correspondence between W-Grammars and Formal Systems of Logic and its Application to Formal Language Description. *Computational Linguistics and Computer Languages XIII* (1979), 19-30.



- [12] Knuth, D.E.: Top-Down Syntax Analysis. *Acta Informatica*, 1, (1971), 79-110.
- [13] Koster, C.H.A.: Affix grammars. *Algol 68 Implementation*, ed. J.E.L. Peck, North-Holland (1971), 95-109.
- [14] Koster, C.H.A.: Two-level grammars. *Lecture notes in computer sciences* 21, *Compiler Construction*, Springer Verlag, Berlin (1974), 140-150.
- [15] Kramer, B., Hein, W., Schmidt, H.W.: On the Implementation of van Wijngaarden Grammars. *IST-Internal Report 3/77*, MBH Bonn, (1977).
- [16] Kramer, B., Schmidt, W.: Locally Nondeterministic and Hybrid Syntax Analyzers from Partition Two-Level Grammars. (1979).
- [17] Marcotty, M., Ledgard, H.F., Rochman, G.V.: A Sampler of Formal Definitions. *Computing Surveys*, 8, 2, 1976, 191-270.
- [18] Maslov, A.N.: Indexed Grammars and van Wijngaarden Grammars. In Russian. *Problem of Information Transfer*, 3/XI (1975), 81-89.
- [19] McKeeman, W.M.: *Compiler Construction*. *Lecture notes in computer sciences* 21, *Compiler Construction*, Springer Verlag, Berlin (1974), 1-36.
- [20] Watt, D.A.: Analysis Oriented Two Level Grammars. Technical University, Berlin, Dep. of Cybernetics Research Group, P2 (1975).
- [21] Wijngaarden, A. van, (ed.): Report on the Algorithmic Language ALGOL – 68. Offprint from *Numerische Mathematik*, Vol. 14, Springer-Verlag (1969), 79-218.
- [22] Wijngaarden, A. van, et al. (eds.): Revised Report on the Algorithmic Language ALGOL-68. Springer-Verlag (1976).

Received 20. June 1980

## REPRESENTATION AND VERIFICATION OF COMMUNICATING SEQUENTIAL PROCESSES

*Tamás Gergely*

Research Institute of Applied Computer Science  
Budapest, Hungary

*Laszló Úry*

Hungarian Central Statistical Office  
Budapest, Hungary

### 1. INTRODUCTION

We aim to develop a verification method for the programming language CSP introduced by Hoare [1]. We propose to build this method on the basis of classical first order logic as to get a tool easy to handle.\*)

At the same time we provide Petri net representation of CSP's texts. This representation gives operational semantics on the one hand while, on the other hand, it makes the program executions visible.

The reader is supposed to be familiar with CSP the notations of which are used without any explanation.

### 2. MAIN FEATURE OF THE PROPOSED METHOD

As it is known CSP program  $P$  is of the form  $[ \mathcal{L}_1 :: p_1 \dots \mathcal{L}_n :: p_n ]$ , where  $\mathcal{L}_i$ 's are the labels and  $p_i$ 's are the constituent processes. A process may contain both I/O commands and I/O guards that, further on, are referred to as I/O constructions. Execution of a process is influenced by that of another one only through the communication prescribed by the I/O constructions. Hence any traditional method can be used to

---

\*) Having had finished this work we learned of Cousot - Cousot [2] and of Hoare [3], who also develop a calculus for CSP programs by the use of ideas slightly similar to ours.



describe the execution of a process  $p_i$  between two communication points. By receiving a message the execution of a process  $p_i$  depends also on the execution of the process sending the message.

Thus it is quite natural to consider the execution of  $p$  such that it is decomposed to parts of the following properties:

- (i) the last element of a part corresponds to the execution of an I/O construction;
- (ii) there is no communication within a part.

As to get this decomposition all I/O constructions and all start and stop points of the processes of  $p$  should be named somehow. A vector  $\alpha = (\alpha_1, \dots, \alpha_n)$  refers to such a state of the program execution in which any process  $p_k$  is going to execute an I/O construction named by  $\alpha_k$  (so  $\alpha$  is called state-vector of  $p$ ). If among these I/O constructions there are pairs (say in  $p_i, p_j$ ) referring to one another then the corresponding communication takes place between  $p_i$  and  $p_j$  and then both  $p_i$  and  $p_j$  are executed up to the next communication point. And that new situation is described by a new-vector  $\beta$ . This fact can be written as  $\alpha, (i \rightarrow j) \mapsto \beta$ . Let us see a simple example where a process labeled by  $N$  calls an adding-routine labeled by  $M$ :

### Example 1

$P =$

$[ N ::$	$\underbrace{M ! a}_{1}$	$;$	$\underbrace{M ! b}_{2}$	$;$	$\underbrace{M ? z}_{3}$	$  $
start						stop

$$M :: \underbrace{N ? x}_1 ; \underbrace{N ? y}_2 ; \Sigma = x + y ; \underbrace{N ! \Sigma}_3 ]$$

start

stop

The possible transitions are:

$(\text{start}, \text{start}) \mapsto (1,1)$   
 $(1,1) (N \rightarrow M) \mapsto (2,2)$   
 $(2,2) (N \rightarrow M) \mapsto (3,3)$   
 $(3,3) (M \rightarrow N) \mapsto (\text{stop}, \text{stop})$

Of course, the transition  $(1,2) (N \rightarrow M) \mapsto (2,3)$  seems to be allowed, but we will prove that the program never reaches the state-vector  $(1,2)$ .

□

The proof of the partial correctness of a CSP program w.r.t. an input condition  $\phi$  and an output condition  $\psi$  can be done as follows.

- (i) For any state-vector  $\alpha$  a formula  $\Phi_\alpha$  should be given to describe the properties of the local program variables in the state corresponding to  $\alpha$  and for any transition say  $\alpha, (i \rightarrow j) \rightarrow \beta$  formulas  $\Theta_{\alpha,i,\beta}, \Theta_{\alpha,j,\beta}$  should be given;
- (ii) Then we have to prove that these formulas are invariant under the possible transitions.

In more detail to any transition  $\alpha, (i \rightarrow j) \mapsto \beta$  well defined parts of programs  $p_i$  and  $p_j$  correspond. By using classical methods the results of the execution of these parts can be described with the help of the formulas  $\Theta_{\alpha,i,\beta}$  and  $\Theta_{\alpha,j,\beta}$ . These formulas describe the properties of the change of the values of program variables under the execution of the corresponding part of processes  $p_i$  and  $p_j$  respectively. The



realization of (ii) supposes the two parts below

- a) It should be proved by the use of any appropriate classical method that the formulas  $\theta_{\alpha, i, \beta}$  and  $\theta_{\alpha, j, \beta}$  are valid.
- b) If the appropriate I/O commands are  $i ! \tau$  and  $j ? x$  then we have to check:  $(\Phi_{\alpha}^* \wedge \theta_{\alpha, i, \beta}^* \wedge \theta_{\alpha, j, \beta}^* \wedge x^* = \tau^*) \Rightarrow \Phi_{\beta}$  local variables of different processes are supposed to be different. The asterix refers to the situation before execution.

For illustration let us see the above program  $p$ .

Example 1 (continuation):

We prove that the program is correct w.r.t. the output condition  $z = a + b$ .

Indeed let

$$\begin{aligned}
 \Phi(\text{start}, \text{start}) & \stackrel{d}{=} \underline{\text{true}} \\
 \Phi(1, 1) & \stackrel{d}{=} \underline{\text{true}} \\
 \Phi(2, 2) & \stackrel{d}{=} a = x \\
 \Phi(3, 3) & \stackrel{d}{=} a = x \wedge b = y \wedge \Sigma = x + y \\
 \Psi \stackrel{d}{=} \Phi(\text{stop}, \text{stop}) & \stackrel{d}{=} z = a + b
 \end{aligned}$$

For other state vectors  $\alpha$  let  $\Phi_{\alpha} = \text{false}$ . It means that the program never reaches the state corresponding to these  $\alpha$ 's, and  $\theta(2, 2), M, (3, 3) \stackrel{d}{=} \Sigma = x + y$ , the other  $\theta$ 's are

$$\theta = (a=a^* \wedge x=x^* \wedge b=b^* \wedge \Sigma = \Sigma^* \wedge y=y^* \wedge z=z^*).$$

Here  $v^*$  denotes the value of the variable  $v$  at the start of the execution of the corresponding part of the program.  $\theta$  says that "nothing has happened".

Now it should be proved that the formulas and  $\theta$ 's are indeed invariant assertions. For this we only check that

$$(\Phi(2,2) \wedge \theta(2,2), M, (3,3))^{\theta} \Rightarrow \Phi(3,3)$$

Namely

$$(a^* = x^* \wedge \theta \wedge \Sigma = x + y \wedge b^* = y^*) \rightarrow (\Sigma = x + y \wedge a = x \wedge b = y)$$

This is obviously true. Verification of the other analogical statement is left to the reader.

□

In order to provide transparent semantics for CSP programs Petri nets are used. A Petri net is an oriented graph with two types of nodes called place and transition respectively.

The construction of the net representation for a CSP program should be carried out in two steps. First the net representation of each constituent process should be constructed such that the transition nodes are labeled by one of the followings:

- (i) by a guard-formula
- (ii) by an assignment
- (iii) by a pair of process labels connected with the sign  $\rightarrow$  and by a message-expression
- (iv) by *end signal* (ES) indexed by a pair of process labels connected with the sign  $\rightarrow$ .

The transitions of the process  $p_i$  labeled as in (iii) correspond to communication, namely a transition node of label  $(i \rightarrow j, \tau)$  in the net corresponding to a process  $p_i$  means that



the process  $p_i$  sends the message  $\tau$  to process  $p_j$ . Those labeled as in (iv) beyond those labeled as in (iii) correspond to the I/O constructions occurring within a repetitive command (signed by \*). Namely  $ES_{i \rightarrow j}$  means that process  $p_i$  will not send messages to process  $p_j$  since the loop of the messagesending process  $p_i$  has already terminated.

The place nodes represent at which point the execution is. Thus the places being right before the transitions labeled either as in (iii) or as in (iv) represent state-vector components. However we label only the places corresponding to start and stop points of the execution.

According to above the net representation for the process A and B in Example 1 look as shown in Fig. 1.

The second step of the net construction of a CSP program is the assembly of the net representation of the constituent process i.e. constructing the resultative net which is called the *bound* one. During this we should remember that a communication takes place between two processes. Thus the construction of the bound net should be done pairwise by binding those transition nodes that correspond to each other in the communicating pair of processes under consideration. Namely, considering say the processes  $p_i$  and  $p_j$  each transition of label  $(i \rightarrow j, \tau)$  of the process  $p_i$  should be bound with each transition of label  $(i \rightarrow j, x)$  of the process  $p_j$ , such that to each possible pair of transition nodes a new transition is rendered. The label of this new node is an assignment which corresponds to the message-expression, i.e. in our case it is  $x := t$ .

Place nodes labeled by *start* of the constituent processes should be replaced by a single place labeled by a vector with *start* components. In a particular case when the CSP program consists only of two processes communicating with each other and in both processes the first command prescribes a communication then the individual *start* place nodes should be bound

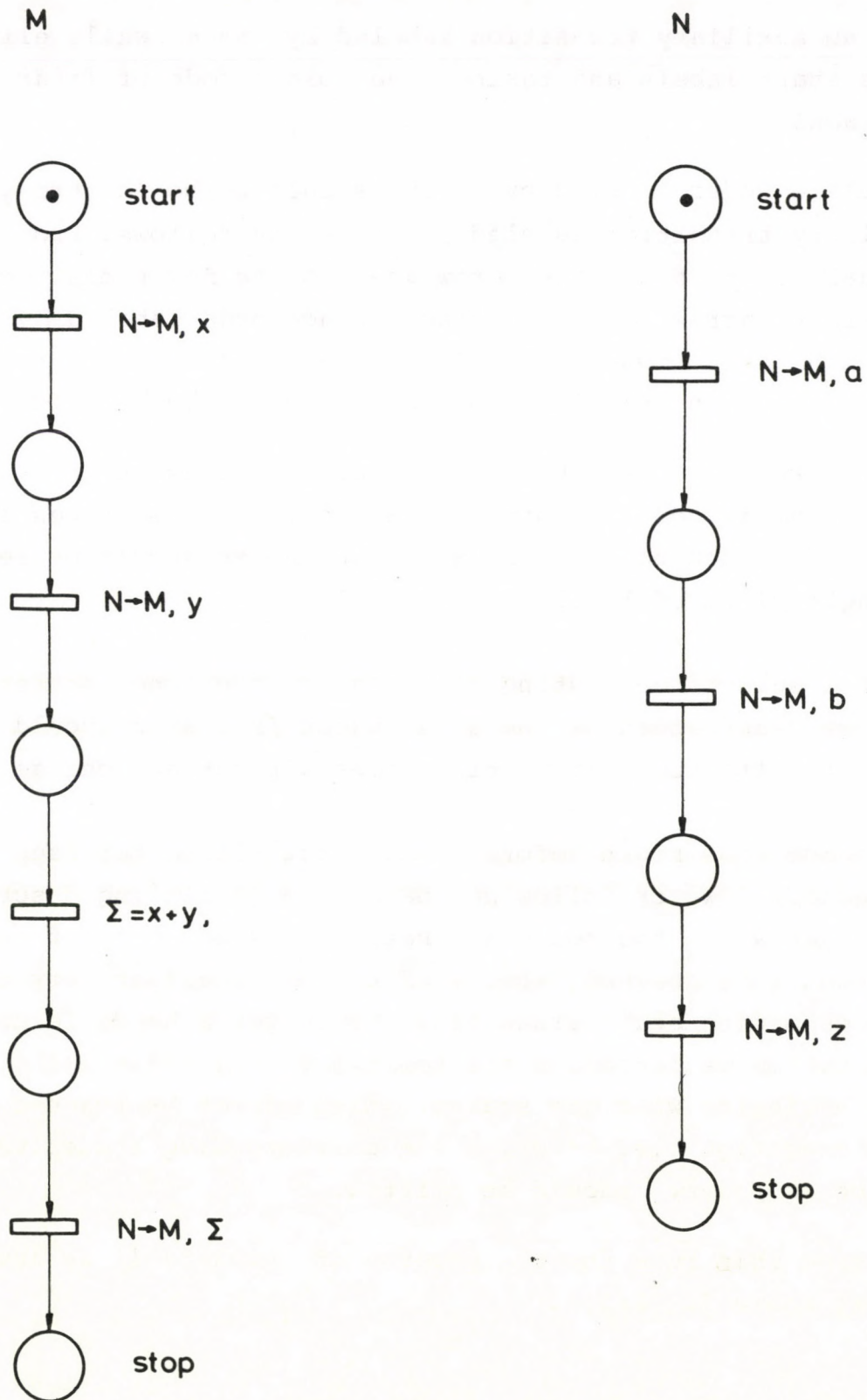


Fig. 1.



through an auxiliary transition labeled by *true* while eliminating the start labels and taking a new place node of (start, start) label.

The place nodes labeled by *stop* should be bound through an auxiliary transition labeled by *true* as follows. From the individual *stop* nodes the arrow goes to the new transition from where an arrow goes to a single place node labeled by a vector with *stop* components. During this construction the *stop* label of the individual processes should be omitted.

In a particular case when the CSP program consists of two processes and in both of them the last command prescribes a communication then the individual *stop* nodes should be replaced by a single place of label (*stop*, *stop*).

All the subnets consisting of nodes not mentioned so far from the net representation of the constituent processes should be hanged onto the bound transition nodes without any change.

The place node right before a bound transition represent a state-vector. However following the method of binding described above we get a far too redundant Petri net since many of the states cannot be reached because of the non realizability of the corresponding state-transition. As to get a handable net representation we introduce the following *reduction rule*: the subnets beginning with the states, which cannot be reached during the execution i.e. to which the corresponding inductive assertion is *false*, should be omitted.

By using this rule for the program of *Example 1* we have the bound net shown in *Fig. 2*.

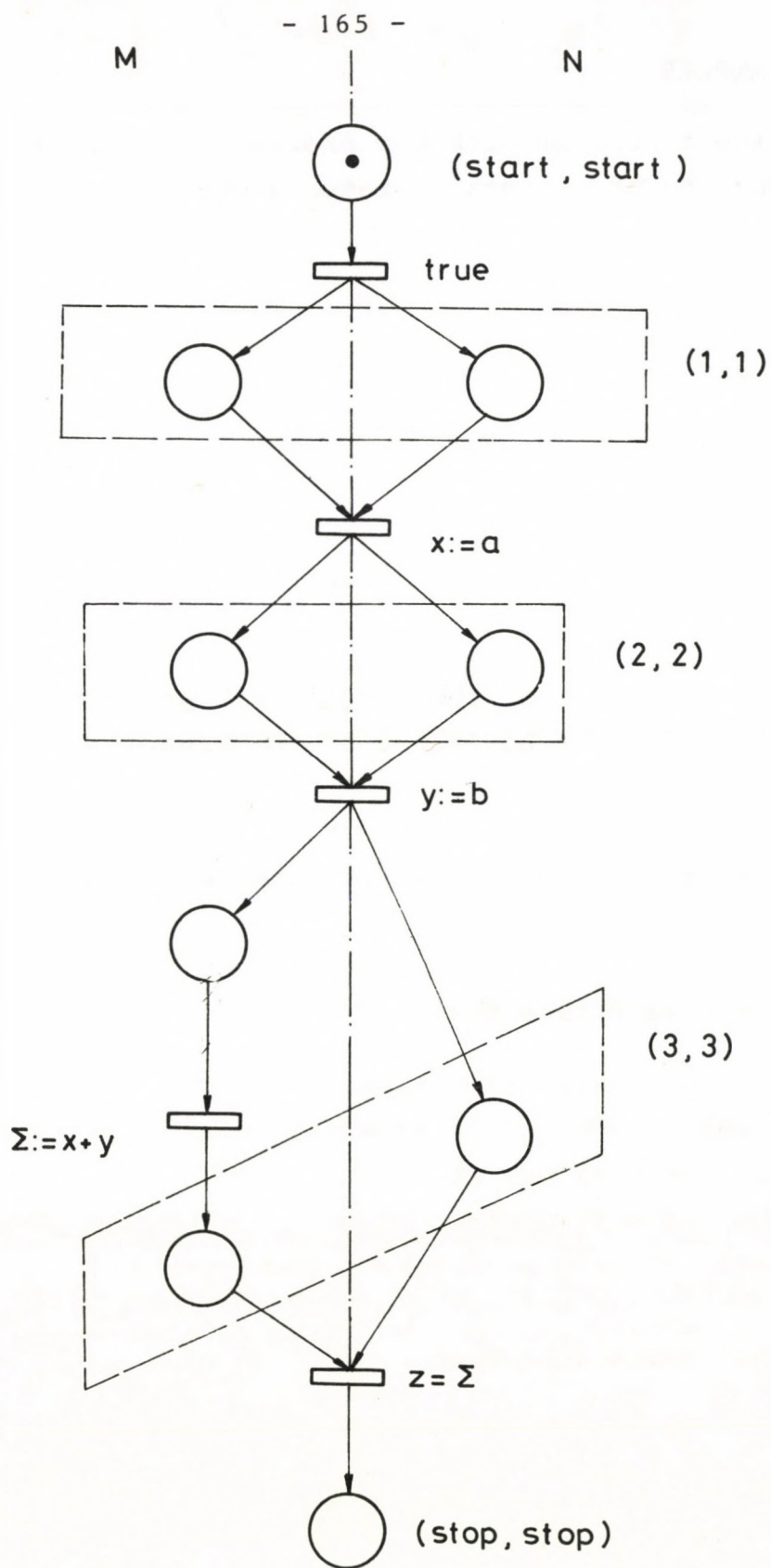


Fig. 2.



### 3. FURTHER EXAMPLES

First we give a program with two processes one of which copies the contents of a stack in reverse order.

#### Example 2

*Copy* =

$$\begin{array}{l} \text{start} \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{stop} \\ [A :: s := x; \overbrace{*[x \neq EOF \rightarrow B!Top(x); x := Tail(x)]}^1] || \\ \\ \text{start} \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{stop} \\ [B :: y : EOF; \overbrace{*[A?z \rightarrow y = Push(y, z)]}^1] \end{array}$$

The net representation of the constituent processes *A* and *B* is shown on *Fig. 3*. The bound net of the program *Copy* is given on *Fig. 4*.

The above program is correct w.r.t. the output condition  $s = y^{-1}$ .

The possible transitions are:

- (i)  $(start, start), (A \rightarrow B) \mapsto (stop, 1)$
- (ii)  $(start, start), (A \rightarrow B) \mapsto (1, 1)$
- (iii)  $(1, 1), (A \rightarrow B) \mapsto (1, 1)$
- (iv)  $(1, 1), (A \rightarrow B) \mapsto (stop, 1)$
- (v)  $(stop, 1), (A \rightarrow B) \mapsto (stop, stop)$

The invariant assertions are:

$$\Phi_{(start, start)} \stackrel{d}{=} \underline{true}$$

$$\Phi_{(1, 1)} \stackrel{d}{=} x.y = s$$

$$\Phi_{(stop, 1)} \stackrel{d}{=} \Phi_{(stop, stop)} \stackrel{d}{=} x.a = s \wedge x = EOF$$

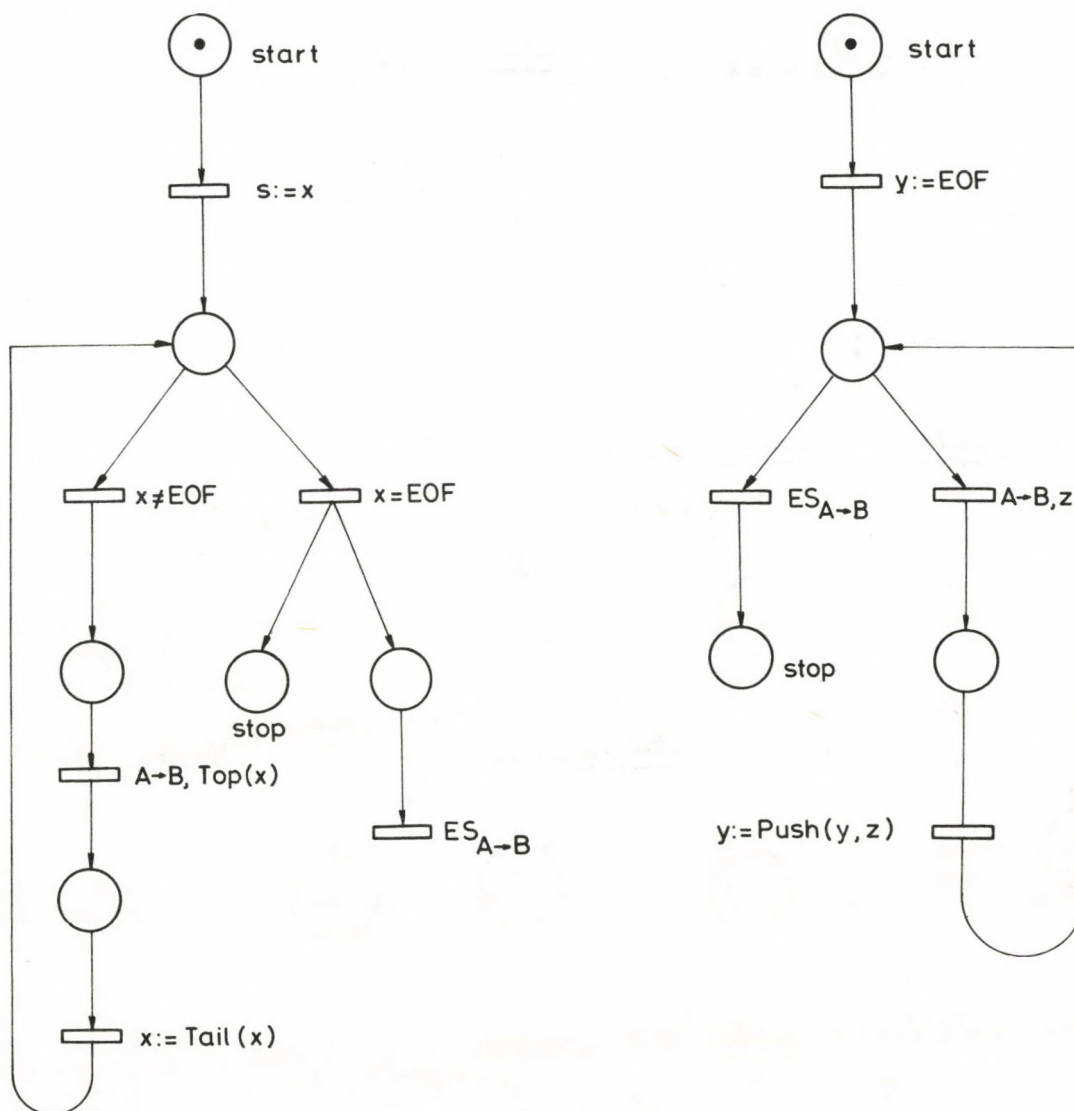


Fig. 3.



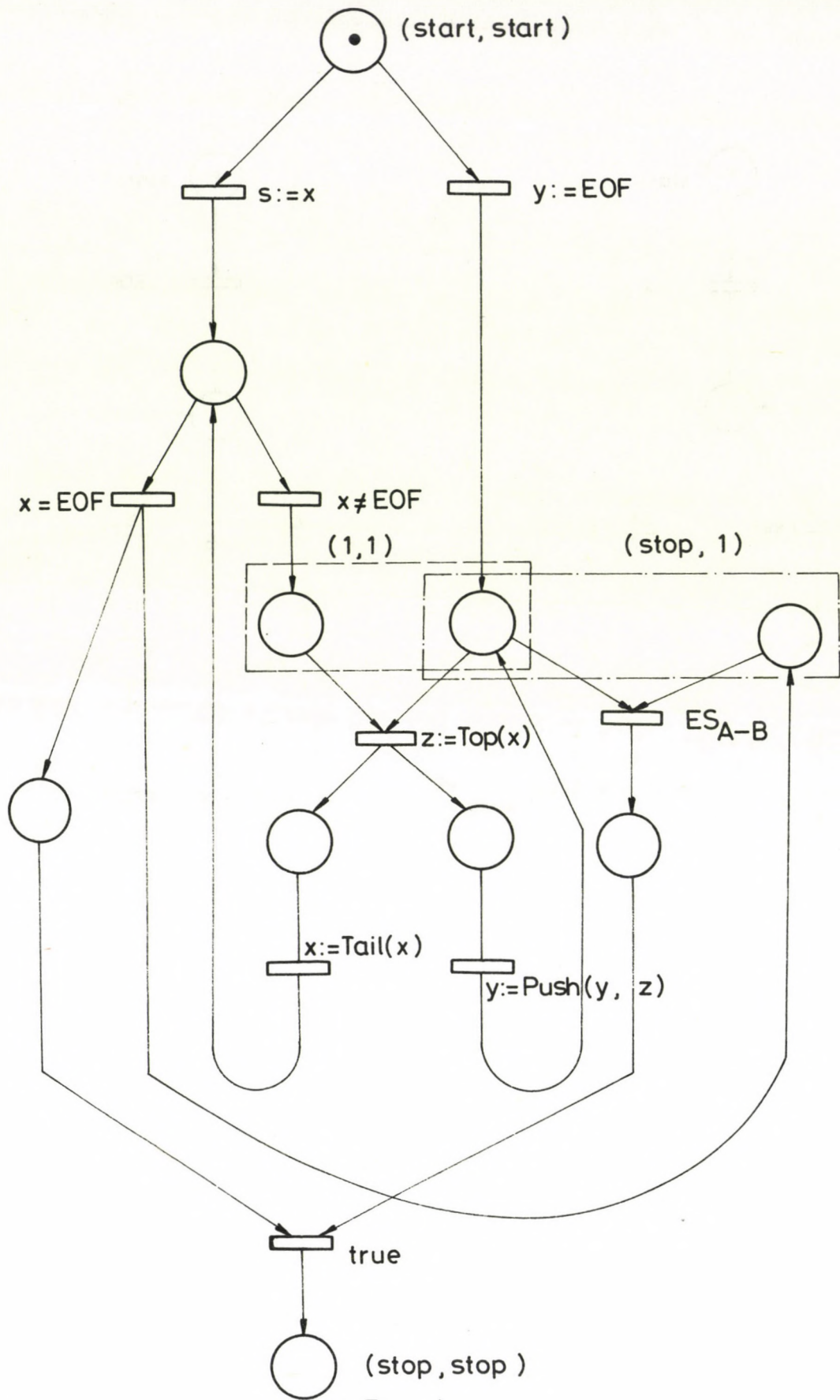


Fig. 4.

$$\theta(start, start), A, (stop, 1) \stackrel{d}{=} s = x \wedge x = EOF$$

$$\theta(start, start), A, (1, 1) \stackrel{d}{=} s = x$$

$$\theta(start, start), B, (1, 1) \stackrel{d}{=} y = EOF$$

$$\theta(start, start), B, (stop, 1) \stackrel{d}{=} y = EOF$$

$$\theta(1, 1), A, (1, 1) \stackrel{d}{=} s = s^* \wedge x = Tail(x^*)$$

$$\theta(1, 1), B, (1, 1) \stackrel{d}{=} z = z^* \wedge y = Push(y^*, z^*)$$

$$\theta(1, 1), A, (stop, 1) \stackrel{d}{=} s = s^* \wedge x = Tail(x^*) \wedge y = EOF \wedge x = EOF$$

$$\theta(1, 1), B, (stop, 1) \stackrel{d}{=} z = z^* \wedge y = Push(y^*, z^*)$$

$$\theta(stop, 1), A, (stop, stop) \stackrel{d}{=} \theta(stop, 1), B, (stop, stop) \stackrel{d}{=}$$

$$s = s^* \wedge x = x^* \wedge y = y^*$$

Now we prove that the above formulas are invariant assertions w.r.t. the possible transitions (i) - (v).

(i) For the transition

$(start, start), (A \rightarrow B) \mapsto (stop, 1)$  we have to prove that  $(s = x \wedge x = EOF \wedge y = EOF) \rightarrow (x.y = s \wedge x = EOF)$  which is trivial.

(ii) For the transition

$(start, start) \mapsto (1, 1)$  we have to prove that  $(s = x \wedge y = EOF) \Rightarrow x.y = s$  which is immediate.



(iii) For the transition

$$(1,1), (A \rightarrow B) \mapsto (1,1):$$

$$\begin{array}{c} \text{old}\Phi \qquad \qquad \text{program A} \\ (x^* . y^* = s^*) \wedge (s = s^* \wedge x = \text{Tail}(x^*)) \wedge \\ \text{program B} \qquad \text{communication} \qquad \text{new}\Phi \\ \hline \wedge (y = \text{Push}(y, z^*) \wedge z = z^*) \wedge z^* = \text{Top}(x^*) \Rightarrow x . y = s \end{array}$$

This is also trivial.

(iv) For the transition

$$(1,1) (A \rightarrow B) \mapsto (\text{stop},1) \quad \text{we have to prove that}$$

$$\begin{array}{c} \text{old}\Phi \qquad \qquad \text{program A} \\ \hline (x^* . y^* = s^*) \wedge (s = s^* \wedge x = \text{Tail}(x^*) \wedge x = \text{EOF}) \wedge \\ \text{program B} \qquad \text{communication} \\ \hline z = z^* \wedge y = \text{Push}(y^*, z^*) \wedge z^* = \text{Top}(x^*) \Rightarrow \\ \text{new}\Phi \\ \hline \Rightarrow x . y = s \wedge x = \text{EOF} \end{array}$$

This is trivial too.

(v) For the transition

$$(\text{stop},1), (A \rightarrow B) \mapsto (\text{stop},\text{stop}) \quad \text{we have that:}$$

$$x^* . y^* = s^* \wedge s = s^* \wedge x = \text{Tail}(x^*) \wedge x = \text{EOF} \wedge$$

$$\wedge y = \text{Push}(y^*, z^*) \wedge$$

$$\wedge z^* = \text{Top}(x^*) \Rightarrow x . y = s \wedge x = \text{EOF}.$$

This is also trivial.

□

Our proof method is powerful enough to prove the correctness of programs with array processes. To show it we give a more complex example, namely a program for printing the prime numbers up to 10 000. Note that here the program is a slight modification of that of Hoare's (see 6.1 of [1]).

### Example 3

*Prim* d

start

A  $\overbrace{[PRINT:: * [i:0...101 ; SIEVE(i) ? m \rightarrow A(m) = \underline{true}]}^1$  stop

B  $\overbrace{||SIEVE(0):: PRINT ! 2; n:=3; * [n<10000 \rightarrow SIEVE(1) ! n]}^1$   $\overbrace{2}$   $n := n+2$  stop

C  $\overbrace{||SIEVE(101):: * [SIEVE(100) ? n \rightarrow PRINT ! n]}^1$   $\overbrace{2}$  stop

D<sub>i</sub>  $\overbrace{||SIEVE i:1...100 :: SIEVE(i-1) ? p; PRINT ! p; mp:=p;}^1$   $\overbrace{2}$

$\overbrace{*[SIEVE i-1 ? m; * [mp < m \rightarrow mp:=mp+p]}^3$

$\overbrace{[mp > m \rightarrow SIEVE (i+1) ! m]}^4$  stop

All possible state-transitions - in a general form - are given in Table 1, where to each transition a row is corresponded. Recall that in the state-vector  $\alpha$  to each process only one component refers. The value of this component refers



Table 1

PRINT	SIEVE (0)	SIEVE (101)	SIEVE (1)		SIEVE (i-1)	SIEVE (i)	SIEVE (i+1)		SIEVE (100)		PRINT	SIEVE (0)	SIEVE (101)	SIEVE (1)		SIEVE (i-1)	SIEVE (i)	SIEVE (i+1)		SIEVE (100)
$\triangle$	$\triangle$	$\triangle$	$\triangle$		$\triangle$	$\triangle$	$\triangle$		$\triangle$		1	1	1	1		1	1	1		1
1	1	1	1		1	1	1		1	$(A \rightarrow B) \rightarrow$	1	2	1	1		1	1	1		1
1	2	.	.		4	1	1		1	$(D_{i-1} \rightarrow D_i) \rightarrow$	1	2	.	.		3	2	1		1
1	2	.	.		3	2	1		1	$(D_i \rightarrow A) \rightarrow$	1	2	.	.		3	3	1		1
1	2	.	.		4	3	.		.	$(D_{i-1} \rightarrow D_i) \rightarrow$	1	2	.	.		3	4			.
1	2	.	.		.	4	3		.	$(D_i \rightarrow D_{i-1}) \rightarrow$	1	2	.	.		3	3	4		.
1	2	2	.		.	.	.		.	$(C \rightarrow A) \rightarrow$	1	2	1	.		.	.	.		.
1	2	.	.		.	.	.		.	$(B \rightarrow A) \rightarrow$	1	$\square$	.	.		.	.	.		.
1	$\square$	.	3		.	.	.		.	$(B \rightarrow D_1) \rightarrow$	1	$\square$	.	$\square$		.	.	.		.
1	$\square$	.	$\square$		$\square$	3	.		.	$(D_{i-1} \rightarrow D_i) \rightarrow$	1	$\square$	.	$\square$		$\square$	$\square$	.		.
1	$\square$	1	$\square$		$\square$	$\square$	$\square$	$\square$	3	$(D_{99} \rightarrow D_{100}) \rightarrow$	1	$\square$	1	$\square$		$\square$	$\square$	$\square$		$\square$
1	$\square$	1	$\square$		$\square$	$\square$	$\square$	$\square$	$\square$	$(D_{100} \rightarrow C) \rightarrow$	1	$\square$	$\square$	$\square$		$\square$	$\square$	$\square$		$\square$
1	$\square$	$\square$	$\square$		$\square$	$\square$	$\square$	$\square$	$\square$	$(D_i \rightarrow A) \rightarrow$	$\square$	$\square$	$\square$	$\square$		$\square$	$\square$	$\square$		$\square$

The sign  $\triangle$  corresponds to start and  $\square$  corresponds to stop

to the name number of the I/O construction to be considered.  
Moreover in the formula  $\Phi_\alpha$  to each element a variable symbol is rendered.

We give only the most interesting assertions, namely:

$$\Phi_{1 \ 2 \ 1 \ \dots \ 4 \ 1 \ 1 \ \dots \ 1} = (\forall j < i) \text{ prim}(j) \nmid m$$

$i-1 \ i$

$$\Phi_{1 \ 2 \ 1 \ \dots \ 3 \ 1 \ \dots \ 1} = p = \text{prim}(i) \wedge m > mp - p$$

$i$

$$\Phi_{1 \ 2 \ 1 \ \dots \ 4 \ 3 \ \dots} = p = \text{prim}(i) \wedge$$

$$(\forall j < i) \text{ prim}(j) \nmid m \wedge$$

$$m > mp - p$$

$$\Phi_{start} = \forall i \quad A[i] = \underline{false}$$

$$\Phi_{stop} = \forall i \quad (A[i] = \underline{true} \longleftrightarrow i \text{ is prime}).$$

One can construct the proof of the correctness of program *prim* without any difficulty.



## REFERENCES

- [1] C.A.R. Hoare., Communicating Sequential Processes,  
Comm. ACM, Vol. 21 (1978), pp. 666-677
- [2] P. Cousot and R. Cousot., Semantic analysis of  
communicating sequential processes,  
Automata, Languages and Programming, Springer-  
-Verlag, LNCS. vol. 85, 1980. pp. 119-133.
- [3] C.A.R. Hoare., A calculus of total correctness for  
communicating processes, Technical Monograph  
PRG-23, Oxford University, 1981.

## SYNTACTIC PATTERN RECOGNITION WITH MODIFIED FUZZY AUTOMATA

*Tibor Gyimóthy and József Dombi*

Research Group on the Theory of Automata

Szeged, Hungary

### Abstract

This paper deals with a syntactic pattern recognition system. Recognition is made by fuzzy automata, input symbols are defined in a fuzzy way. Input symbols are generated by a fuzzy automaton, controlled by aid of a corrective errortable.

### 1. Introduction

In this decade with respect to the general use of computer evaluating systems the problem of pattern recognition became more important.

To identify and classify symbols there are two totally different procedures, one based on statistic methods, the other the syntactic pattern recognition.

The main purpose of statistic pattern recognition is to reduce the probability of wrong classifying to a minimum. The method can be applied on condition that every attribute is given by a vector of the length  $n$  ( $n$  is fixed). In the recognition algorithm each component of the vector has the same significance.

The classes to be recognized during syntactic recognition are defined by grammars. Pattern representation is given by a sentence-like form. Recognition is made by the parsers of the above mentioned grammars and classification is given by the class belonging to the grammar accepting the sentence.

Another condition of this application method is that the grammars should be selective as far as the possible patterns are concerned.

Both procedures can be applied in certain cases well and in other cases not. E.g. the statistic method can be used successfully for analysing EEG curves [1] whereas syntactic pattern recognition is used for classifying chromosomes [2].

Generally, the following statement can be made.

"When patterns are very rich in structural information and the recognition problem requires classification and description, then syntactic approach seems necessary" [3].

Handwritten letters have not been recognized successfully, by neither of both methods. The systems used at present are the recognition of handwritten letters based upon heuristic processes. The trouble in recognizing these letters is that their type may change when their



structure, size or direction of certain elements is somewhat modified. We can say that the type of letters remains unchanged by certain transformations, whereas other transformations change it in spite of the fact according to which the modification of the original pattern caused by the two transformations is of the same degree.

I.e. the different parts of the letters are not of equal significance (see Figure 1). These different significances can be treated efficiently by the fuzzy syntactic pattern recognition systems.

Our purpose was to construct a system working on fuzzy data yielding fuzzy results by the aid of classes, defined by fuzzy grammar. The fuzzy results show the pertaining magnitude of the classes for figure recognition.

The reason why we have chosen the examination of handwritten letters was:

(i) No special knowledge is needed for evaluation of the results (as e.g. in the case of ECG curves).

(ii) The letters' structure, their inner proportions, the direction of the different elements is not unambiguously determined. (The fuzzy writing of handwritten letters is the normal writing process, since the learning of letter-writing is nothing else than executing fuzzy commands.)

(iii) The problem is not solved satisfactorily.

We tried to construct the system for general purpose, as much as possible, i.e. by changing the fuzzy grammars, the system will be able to solve other pattern recognition problems too.

## 2. Formal Description of the System

In this part we give the formal description of system. The sets and mappings mentioned here will be fully expounded in Chapter 3. It is necessary to note that we do not give any precise mathematical description, we formalize only for the sake of uniform treatment.

Let us take a pattern (see e.g. Figure 2) denote it by  $M$ . and divide it into pieces. (E.g. in a way that there should be no intersection of the resultant parts, see: Figure 3) Let us call the resultant parts: segments.

The set of the segments will be denoted by

$$(1) \quad A = \{a_1, \dots, a_n\}$$

Let

$$(2) \quad P = \{p_1, \dots, p_k\}$$

be the set of those attributes which characterize the segments (see e.g. Figure 4).

Let

$$(3) \quad S = (V_0^+)^k$$

where  $V_0^+$  denotes the non-negative real numbers.

Let us introduce a mapping  $\delta$

$$\delta : A \rightarrow S$$

(4)

$$\delta(a_i) = \langle s_{i_1}, \dots, s_{i_k} \rangle$$

where  $s_{i_m}$  shows to what extent  $a_i$  has the  $p_m$ -th attribute.

We denote by

$$(5) \quad D = \{d_1, \dots, d_t\}$$

the set of structural symbols, i.e. the set of those elements with the use of which the structural connections between the segments can be described.

Let

$$(6) \quad T = (V_0^+)^t$$

and  $\gamma$  a correspondence of  $D$  and  $T$ .

$$(7) \quad \gamma : D \rightarrow T$$

where  $\tau_{i_m}$  shows for  $\langle \tau_{i_1}, \dots, \tau_{i_t} \rangle \in T$  with what certainty the structural symbol  $d_m$  can be applied. ( $1 \leq m \leq t$ )  $\gamma$  is a correspondence, because the same structural symbol can be applied at different places with different certainty.



Let

$B = A \cup D$  and  $S' = S \times T$ , and  $\varphi$  a correspondence of  $B$  and  $S'$ , that is

$$\varphi : B \rightarrow S'$$

Where

$$\varphi(a) = \langle \delta(a), \overbrace{0,0,\dots,0}^t \rangle \quad \text{if } a \in A$$

and

$$\varphi(a) = \langle \underbrace{0,0,\dots,0}_k, \gamma(a) \rangle \quad \text{if } a \in D$$

Let  $B' = \varphi(B)$  and let  $F(B')$  denote the free monoid generated by  $B'$ . A  $C$  element of this yields the fuzzy description of pattern  $M$ . That is,  $C$  consists of ordered  $\langle C_{i_1}, \dots, C_{i_{k+t}} \rangle$  tuples, in which if  $1 \leq m \leq k$  then  $C_{i_m}$  shows to what extent the  $p_m$ -th attribute is realized; if  $k < m \leq k+t$  then  $C_{i_m}$  shows the applicability of the  $(m-k)$ -th structural symbol.

We define the  $A = \{E, H, Y, E_0, \omega, \beta\}$  fuzzy automation where  $E$  is the set of states,  $H$  is the set of input symbols,  $Y$  is the set of output symbols,  $E_0$  the initial state,  $\omega$  the transition function,  $\beta$  the output function.

Let

$$E = \{E_0, \dots, E_s\}$$

where

$E_i$  is of the following form:

$$(9) \quad E_i = \begin{bmatrix} e_{i_1} & \dots & e_{i_{k+t}} \\ f_{i_1} & \dots & f_{i_{k+t}} \end{bmatrix}$$

This means, that the elements of  $E$  are matrices of dimension  $2 \times (k+t)$ , where  $e_{i_m}$  is a weight showing the applicability of the  $m$ -th input in the  $i$ -th state ( $e_{i_m} \geq 0$ ), and  $f_{i_m}$  denotes the new state after the arrival of the  $m$ -th input symbol.

$$(0 < f_{i_m} \leq 1, 1 \leq m \leq k + t)$$

Let

$$H = (V_0^+)^{k+t}$$

If precision and magnitude of real numbers are limited, then  $H$  is finite. The transition function  $\omega$  maps  $ExH$  into  $E$ .

Let us suppose that automaton  $A$  is in state  $E_i$  (9) and the input signal  $H_j = \langle h_{j_1}, \dots, h_{j_{k+t}} \rangle$  arrives.

Let

$$(10) \quad b = \max_{m=1, \dots, (k+t)} (\min(e_{i_m}, h_{j_m}))$$

and let  $m'$  be the value of  $m$  at this maximum. If the value of  $b$  can be obtained to be more than one  $m$ , then let us choose the first  $m$  in succession for  $m'$ .

Now the transition function  $\omega$  can be defined as:

$$\omega(E_i, H) = f_{i_{m'}}.$$

Let  $Y = V_0^+$  and  $F(Y)$  is the free monoid, generated by  $Y$ . The output function  $\beta : ExH \rightarrow Y$  can be defined as

$$\beta(E_i, H_j) = b$$

Let us extend the domain of  $\omega$  and  $\beta$  to  $ExF(H)$ , where  $F(H)$  is the free monoid, generated by  $H$ .

Let  $G \in F(H)$  be an optional word and  $I \in F(Y)$  the image of  $G$  at the mapping, induced by automaton  $A$ .

In this case we can say that automaton  $A$  recognizes the word  $G$  with a certainty  $\psi(I)$ , where

$$\psi : F(Y) \rightarrow [0,1].$$

The function  $\psi$  can be defined as follows:

Let  $Y_i = \langle y_{i_1}, \dots, y_{i_j} \rangle \in F(Y)$  optional, then

$$\psi(Y_i) = \min_{m=1, \dots, j} (y_{i_m}).$$



### 3. Description of the System

The system gets a matrix of dimension  $20 \times 20$  as an input. Let us denote it by  $M$ . 1 means a 'dark' cell, 0 the empty one. We work with 'noisy' figures (see Figure 2), so a preliminary filtering is required. As a first step we make *chains* from the elements of the pattern. With few modification we make it in a way suggested by Freeman [5]. We differentiate eight directions (see Figure 5) and we always start in the direction being nearest to the previous one. In Figure 6/a we can see how Figure 2 looks after making a chain. The identical elements belong to one chain, the coding of the chains is shown in Figure 6/b. according to their direction. Now we make one-layer chains, i.e. we obliterate the *part-chains* which increase the thickness of only one chain. We make chains from the residual elements again (see Figure 6/c).

We note that we reduce the neighbouring nodes into one, in order to make our system more effective.

The nodes yield a natural segmentation of chains. These segments will be the elements of set A. It should be noted here, that set A can be extended, because if a segment of A cannot be characterized properly, this segment will be divided into two parts at its greatest fracture, there we take there as members of A instead of the original one. It is easy to find the greatest fracture by the aid of the direction-coded form of the chains.

Set P(2) consists of 10 elements in our system. The description of the elements of P is contained in Figure 4. The  $\delta$  mapping is realized by a fuzzy automaton, controlled by a corrective error-table. Error-table and description of the "lower curve" automaton is shown in Figure 7. The input symbols of the automaton are the direction-coded forms of the segments.

If the symbol arriving at the automation is able to *tip it over*, we do not count an error. If the automaton is in state  $S_i$  and the arriving  $C_j$  ( $C_j = 1, \dots, 9$ ) symbol switches it *over into an error state*, we have to carry out the following:

Let ET denote the error table, and  $C_{j_1}, \dots, C_{j_k}$  the input symbols by whose help we can step further from state  $S_i$ . Let us form the expression

$$s = \min_{t=1, k} ET(C_j, C_{j_t})$$

Let  $t'$  denote the value of  $t$  where this minimum is given. In this case the automaton is tipped over from state  $S_i$  according to symbol  $C_{j_{t'}}$ , and the product  $s \cdot f_i$  is added to the error of the deduction, where  $f_i$  is the fuzzy weight ordered to  $S_i$ . (We note that to pertains to one  $f_i$ ).



Let  $a_q$  be the segments which we want to characterize, and let  $e_m$  denote the deduction error of the  $m$ -th automaton, at the analysis of  $a_q$ .

Applying mapping  $\delta$  to  $a_q$ , we get:

$$\delta(a_q) = \langle s_{i_1}, \dots, s_{i_k} \rangle,$$

where

$$s_{i_m} = g(e_m/l(a_q)) \quad (1 \leq m \leq k).$$

$l(a_q)$  denotes the length of the segment  $a_q$ ,  $g(x)$  is a properly chosen function. To increase effectiveness of our system we introduce a *cutting point*. If the deduction error in the  $m$ -th automaton is greater than  $\eta$ , automaton finishes parsing and  $s_{i_m} = 0$ . Figure 8 we can see a segment, its direction-coding and the images of the segments at the mapping  $\delta$ .

The structural description of the pattern is given by a PDL-like expression [6]. The set  $D(5)$  consists of structural symbols necessary for the PDL expression

$$D = \{+, x, \mathfrak{A}, (, )\}$$

Note that we do not use operation  $-$  suggested by Shaw [6].

In our system uncertainty of structural symbols is not allowed, i.e. if  $\tau_{i_m} > 0$ ,  

$$\tau_{i_j} = 0 \quad \begin{matrix} 1 \leq j, m \leq t \\ j \neq m \end{matrix}$$

for every  $i$  in  $T(6)$

Thus (7) is a very simple mapping in our case and might become much more sophisticated in other systems.

In Figure 9 the PDL-like formulation of Figure 3. can be seen a form corresponding to  $C$  in our formal description.

This form yields the input of fuzzy recognizing automata.

#### 4. Description of the Recognizing Automaton

Up to now we have examined the recognizing system in general. Further on we construct special automata which recognize block letters. In order to achieve more effective generalized performance further comparison is made with the formal description.



For the choice of the letter-types we have used the patterns of the paper written by S. H. LU and K.S. FU [7] and have applied the recognizing algorithm to their letter, too. We have examined nine types of letters altogether.

These symbols cannot be separated easily. If their structure, size and direction of their elements are somewhat changed their type may change, too. (see Figure 10).

Since the type of the letters does not change when the letters are lengthened or shortened horizontally or vertically, it seemed to be practical to supplement the fuzzy input of our automaton with those horizontal and vertical proportions which compare the total size of the letters with the size of the letter-segments.

Further on the type of letters will be invariable to turning by a small angle, since the writing patterns show that the slanting way of writing is rather common. We have modified the automaton's performance so that it recognizes slanting letters as well as straight ones.

Now, let us describe the automaton recognizing a given letter and the way how it works. Let us see the possible forms of a letter (see Figure 11) and the PDL-like expressions belonging to them.

- 1.)  $a_1 x(c_1 x a_2 + a_3) + a_4$
- 2.)  $a_1 x(c_1 x a_2 + a_3) x a_4 + d_1$
- 3.)  $a_1 x(c_1 x a_2 x d_2 + a_3) x a_4 + d_1$
- 4.)  $a_1 x(c_1 x a_2 x d_2 + a_3) x a_4 + d_1$

The possible values of segment types are:

$$\begin{aligned} a_i &= \backslash \mid / \\ c_i &= \backslash - / \\ d_i &= \backslash - / G \end{aligned}$$

where G denotes those elements which cannot belong to any segment type because of the shortness of the segment. To each segment identifying symbol of the PDL descriptions (1 - 4) we order the horizontal and vertical proportions (h,v) of the segment.

For the acceptance of the segment identifying symbols by our automaton it is necessary that the corresponding (h,v) values be the elements of a certain fuzzy subset of the [0,1] interval.

This corresponds to the fuzzy description according to which the element  $c_1$  of H is in the middle.

During the description of the letters we use 10 fuzzy subsets altogether. (see Figure 12). Now, let the automaton in the j-th state, and suppose that i-th input symbol arrives. Then



the automaton orders the value

$$R_i = \max_k \min (c_{i_k}, e_{j_k}, F_h(h_i), F_v(v_i))$$

to this symbol, where  $c_{i_k}$  and  $e_{j_k}$  are defined in Section 2.  $F_v$  and  $F_h$  are the appropriate vertical and horizontal functions, and  $(h_i, v_i)$  the horizontal and vertical proportions of the symbol. If the maximum is  $k'$  the automaton switches into state  $f_{j_{k'}}$ , (see Section 2). If  $R_i < \alpha$  ( $\alpha$  a cuttingpoint) then the letter is not accepted by the automaton. In the other case the letter is recognized with certainty

$$\min_i R_i.$$

The fuzzy automaton constructed in this way can be easily adapted to a given letter-type by modification of the fuzzy function system.

## Conclusions and Notes

Our system was implemented on a CDC-3300 computer. The average time of recognition of a pattern is 2 sec. We note that this time can be considerably reduced if we give the cutting-point a fairly large value.

Our system recognized 93 percents of the patterns properly, 2 percents with a mistake and 5 percents of the patterns were not classified. Our experiences show that the main source of the mistakes is that the same pattern can be written in several ways in PDL.

The process of learning can be interpreted as the optimal setting of the fuzzy weights of the automaton. We continue developing our system in this direction.

## References

- [1] T. Young and T.W. Calwert, Classification and Estimation Pattern Recognition (Elsevier)
- [2] E.T. Lee, Shape-Oriented Chromosome Classification, IEEE Transaction on System, Man and Cybernetics Nov. 1975.
- [3] K.S. Fu, Syntactic Methods in Pattern Recognition, Academic Press, New York (1974)
- [4] K. Tanaka and M. Mizumoto, Fuzzy Programs and their Execution, Fuzzy Sets and their Applications to Cogniture and Decision Processes, Editor by L. A. Zadeh, K.S. Fu, K. Tanaka: Academic Press.





Figure 1

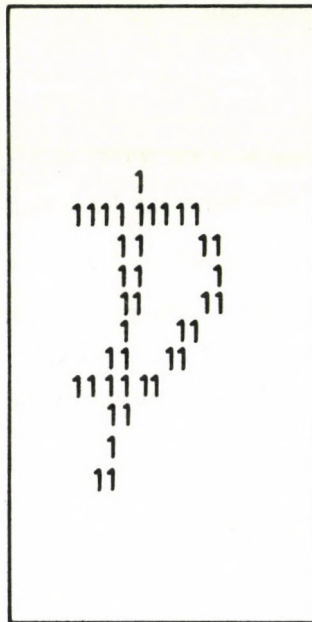


Figure 2





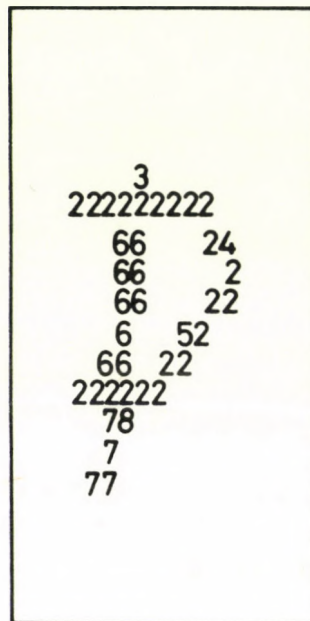


Figure 6/a

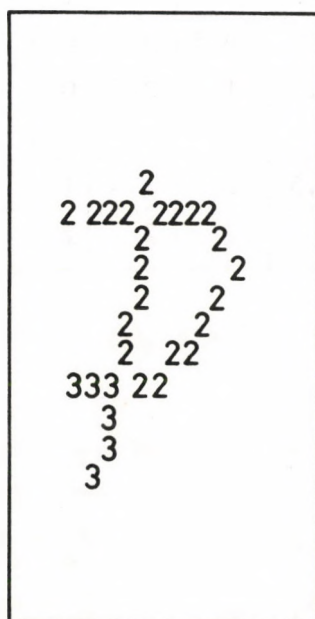


Figure 6/C



	1	2	3	4	5	6	7	8	9
1	0	0.5	1	1.5	2	1.5	1	0.5	2
2	0.5	0	0.5	1	1.5	2	1.5	1	2
3	1	0.5	0	0.5	1	1.5	2	1.5	2
4	1.5	1	0.5	0	0.5	1	1.5	2	2
5	2	1.5	1	0.5	0	0.5	1	1.5	2
6	1.5	2	1.5	1	0.5	0	0.5	1	2
7	1	1.5	2	1.5	1	0.5	0	0.5	2
8	0.5	1	1.5	2	1.5	1	0.5	0	2
9	2	2	2	2	2	2	2	2	2

$$S_1 \rightarrow 1S_2 \quad 2$$

$$S_5 \rightarrow 3S_5 \quad 4$$

$$S_1 \rightarrow 8S_3 \quad 2$$

$$S_5 \rightarrow 4S_6 \quad 4$$

$$S_1 \rightarrow 2S_4 \quad 1$$

$$S_5 \rightarrow 5S_7 \quad 4$$

$$S_2 \rightarrow 1S_2 \quad 1$$

$$S_6 \rightarrow 5S_7 \quad 1$$

$$S_2 \rightarrow 2S_4 \quad 1$$

$$S_6 \rightarrow 4S_6 \quad 1$$

$$S_2 \rightarrow 3S_5 \quad 1$$

$$S_6 \rightarrow 9S_{10} \quad 1$$

$$S_3 \rightarrow 1S_2 \quad 1$$

$$S_7 \rightarrow 9S_{10} \quad 2$$

$$S_3 \rightarrow 8S_3 \quad 1$$

$$S_7 \rightarrow 9S_{10} \quad 2$$

$$S_4 \rightarrow 1S_2 \quad 1$$

$$S_4 \rightarrow 2S_4 \quad 1$$

$$S_4 \rightarrow 3S_5 \quad 1$$

where "9" is the terminal symbol  
and "S<sub>10</sub>" the final state.

Fig. 7

Chains

- 2. ) 333333332217887877777
- 3. ) -
- 4. ) -
- 5. ) -
- 6. ) 35555311
- 7. ) 355
- 8. ) -

Figure 6/b





V V, Y Y, X X,  
K F, K H, P P

Figure 10

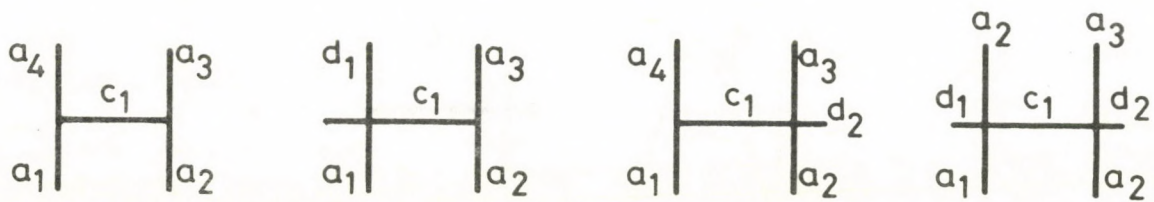


Figure 11



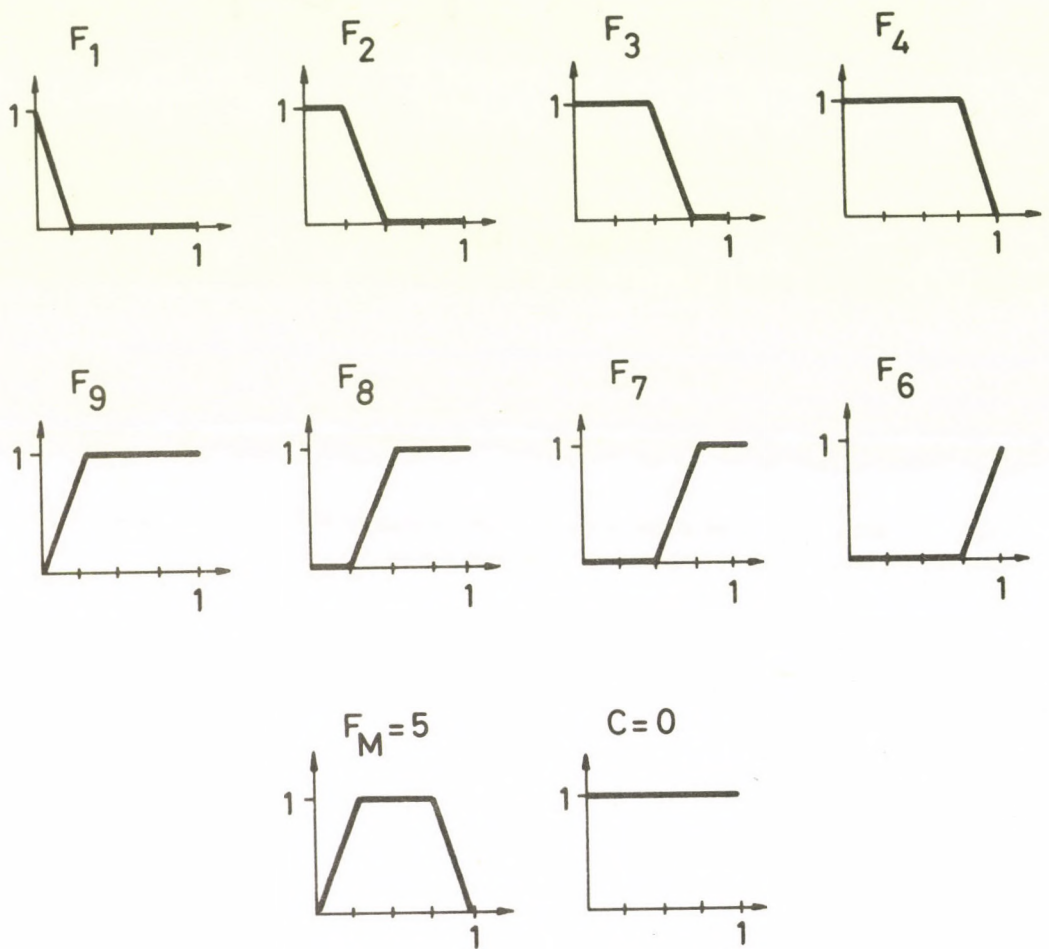


Figure 12

- [5] H. Freeman, IEE Trans EC-10, 260 (1961)
- [6] A.C. Shaw, Parsing of Graph Representable Pictures, JACM Vol , 7. No 3. July , 1970; pp 453-481
- [7] S.H. LU and K. S. Fu, A Sentence-to-Sentence Clustering Procedure for Pattern Analysis, IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-8, No 5. May 1978.





## MORPHOLOGICAL AND MORPHONOLOGICAL ANALYSIS OF HUNGARIAN WORD-FORMS BY COMPUTER

*Gábor Prószéky – Zoltán Kiss – Lajos Tóth*

ELTE University  
Budapest, Hungary

### 1. INTRODUCTION

The computational syntactical analysis of Hungarian sentences with any purpose, is impossible without accurate examination of the words (word-forms) occurring in the sentence, since only the connections between the stem and the affixes can define the surface structure of the sentence. Of course there is another, and perhaps more significant component from the point of view of the recognition of new information: word-order, but it is not influenced by the description of the surface structure of the sentence. We shall have an independent task which follows the task of the syntactical analysis and this task deals with word-order.

Therefore, if we want to describe the surface structure by a tree, it is necessary to define the syntactical functions of several words and word-groups and to detach the affixes carrying this information from the word that makes one word-form with this affix according to the rules of the Hungarian language.

This step is a real morphological analysis (in the first approximation). The problem is caused by the rule of affix-connecting, based on phonological junctions of the Hungarian language. It is a complex system of rules which is so evident for people whose mother tongue is Hungarian that they must spend almost no time on watching it functioning when a sentence is heard or read. But that is not the case if we "understand" the syntactical structure of an ambiguous or a rather complicated sentence; and even the recognition of errors originating from the incorrect use of the rule system is carried out more



intuitively than the recognition of errors of sentence-constructing. It seems that "linguistic sensitivity" of common people whose mother tongue is Hungarian is finer than their "musical sensitivity".

There are two fundamental problems connected with the "false" note:

- (1) Can we suppose that the "text generator" does not send out "false notes"?
- (2) If the "text generator" does, can we determine the behaviour of the perceptor in this case?

If we answer "yes" to question (1), then we defined an automaton, called strong automaton. It contains a system of rules which can be used by the automaton without errors.

Consider case (2). The answer "yes" to this question means the following: the automaton defined in this way recognizes the "correct notes", but its behaviour is undetermined in the case of the "false note". We call it weak automaton. The answer "no" to case (2) includes the fact that this behaviour is not the direct continuation of the analysis but the generation of the question for the sake of correcting the "false note". The automaton received will be called: correcting automaton.

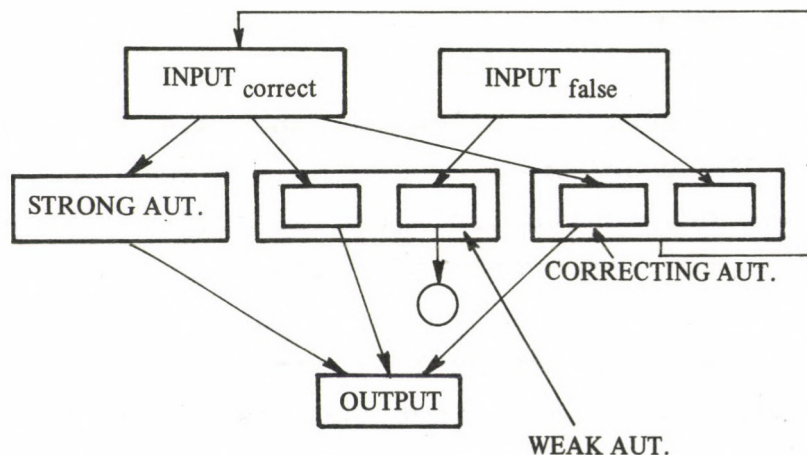


Fig. 1.1



We are going to deal with the strong automaton, namely we suppose to analyse only proper Hungarian texts. (Thus our model does not serve the purpose of exercising Hungarian language).

Later, we shall augment our recognizing model; of course, not only dealing with the "false note" can cause us great anxiety, but the next problem, too. We can briefly summarize the functioning of our automaton in the following manner: the automaton must verify whether the sequences of the elements it knows are edited according to the rules it knows, or not. Incompatibility with the rules can then be realized on the level of morphonology and on the level of morphology (this is the "false note" problem).

Augmentation of the automaton will be necessary if the input contains elements not known by the original automaton. In these cases the augmented automaton will construct particular alternatives by consistent application of the rules. Then it will generate a question and will complete the analysis with the help of the answer it has got. We call this automaton an inferential automaton. (Properly speaking it is a "learning" automaton, because it takes reference to the new recognized element, and it treats this element as the originally known one. But we did not want to use the expression "learning automaton", since it is reserved for another concept.) The subclasses of the inferential automaton are: strong inferential automaton, weak inferential automaton and correcting inferential automaton. The automaton we review here is a strong inferential automaton.

## 2. BASIC CONCEPTS OF AUTOMATICAL ANALYSIS

Since we deal with automatical language processing, we cannot forget that all components of the system which we want to realize in practice, are finite ones, i.e.: there are bounds which are debatable from the point of view of linguistics, but for realizing the system the problem needs a concrete attitude.



The problematic finite components are:

- (a) the vocabulary of stems which is finite, but can increase during the analysis (in the case of an inferential automaton).

Questions: - How many elements it will have?

- What do we consider a stem?

(b) the vocabulary of the affixes which is a simple set of affixes. There are no operations defined on this set, thus the concatenations of affixes can be in the input strings only. In the vocabulary of affixes there are only the atomic elements of this string. Seemingly the complex verbal affixes contradict this. These complex affixes are real compositions and not simple affixes from the linguistic point of view, but we can treat them as simple elements of the vocabulary of affixes. There are practical considerations forcing us to use these complex affixes, namely the diachronic change left ripples principally here in the Hungarian language, which can be said an agglutinating one in the cases of nominals.

We note also that the set of affixes has no subclasses, that is, the variants too are equal elements of the set of affixes.

Question: - Which affixes do we conclude as old-fashioned or unproductive, excluding them from further examinations?

As we can consider further, the vocabulary of stems and affixes is finite, and by the set of the morphological and morphonological rules we try to outline the task more exactly with the help of the initiation of some new notions. These notions denote abstract grammatical elements and they will occur in many places of the following chapters. We will treat the definitions and the properties of the concrete morphological and morphonological rules in Chapter 5.3.



### 3. EXPLANATION OF THE DESIGNATIONS

#### 3.1 Stems

VERB	verb	}	NOM	nominal
SUBS	substantive			
ADJ	adjective			
NUM	numeral			
PRON	pronoun			
ADV	adverb			
MOD	modifying word			
VPR	verbal prefix			
ART	article (D: definite, I: indefinite)			
PPS	postposition			
INT	interjection			
CON	conjunction			

#### 3.2 Verbal affixes

(a) - Having a form of 5-tuple

(The points denote the other four positions)

S....	subjective(indefinite)	}	<u>type of conjugation</u>
O....	objective(definite)		
.D...	declarative(indicative)	}	<u>mood</u>
.C...	conditional		
.S...	subjunctive		
..Pr..	present	}	<u>tense</u>
..Pa..	past		
...1.	singular	}	<u>number</u>
...2.	plural		
....1	first	}	<u>person</u>
....2	second		
....3	third		



Their possible compositions:

SDPr..      ODPr..      SDPa..      ODPa..  
SCPr..      OCPr..  
SSPr..      OSPr..

(To the places of the points: 11, 12, 13, 21, 22, 23)

(Note: Hungarian has no affixes for the future tense. It is replaced by the present form plus an adverb, or a composition of a verb the auxiliary verb fog.)

(b) - Gerund with a personal suffix (properly speaking it is the conjugation of the infinitive)

GER..      gerund with a personal suffix

(To the places of the points: 11, 12, 13, 21, 22, 23)

(c) - Other verbal affixes (without any number and person)

<u>abbr.</u>	<u>type of the affix</u>	<u>morphemes</u>
INF	infinitive	<u>ni, ani, eni</u>
VN	verbal noun	<u>ás, és</u>
CPT	continuous participle	<u>ó, ő</u>
PPT	perfect participle	<u>t, tt, att, ett,</u> <u>ott, ött</u>
FPT	future participle	<u>andó, endő</u>
APS	adverbial participle(simult.)	<u>va, ve</u>
APA	adverbial participle(anteced.)	<u>ván, vén</u>

### 3.3 Nominal affixes

<u>abbr.</u>	<u>type of the affix</u>	<u>morphemes</u>
ABL	ablative	<u>tól, től</u>
ACC	accusative	<u>t, at, et, ot, öt</u>
ADE	adessive	<u>nál, nél</u>
ALL	allative	<u>hoz, hez, höz</u>
CAU	causative	<u>ért</u>
COMP	comparative	<u>bb, abb, ebb, obb</u>
DAT	dative	<u>nak, nek</u>



<u>abbr.</u>	<u>type of the affix</u>	<u>morphemes</u>
DAY	day(used in expressing the date)	<u>adikán</u> , <u>edikén</u> , <u>odikán</u> , <u>ödikén</u>
DEL	delative	<u>ról</u> , <u>ről</u>
DIS	distributive	<u>nként</u> , <u>anként</u> , <u>enként</u> <u>önként</u> , <u>önként</u>
DIS-TEM	distributive-temporalis	<u>nta</u> , <u>nte</u> , <u>anta</u> , <u>ente</u> , <u>onta</u> , <u>önte</u>
ELA	elative	<u>ból</u> , <u>ből</u>
ESS	essive	<u>ul</u> , <u>ül</u>
FAC	factivitive	<u>vá</u> , <u>vé</u> , <u>*á</u> , <u>*é</u>
FOR <sup>1</sup>	formalis	<u>ként</u>
FOR <sup>2</sup>	formalis	<u>képp</u> , <u>képpen</u>
FRA	fraction numeral	<u>d</u> , <u>od</u> , <u>ed</u> , <u>öd</u>
I	- <u>i</u> (derivative ending)	<u>i</u>
IK	- <u>ik</u> (formative syllable)	<u>ik</u>
ILL	illative	<u>ba</u> , <u>be</u>
INE	inessive	<u>ban</u> , <u>ben</u>
INS	instrumentalis	<u>val</u> , <u>vel</u> , <u>*al</u> , <u>*el</u>
LOC	locative	<u>t</u> , <u>ott</u> , <u>ett</u> , <u>ött</u>
MOD-ESS <sup>1</sup>	modalis-essive	<u>ul</u> , <u>ül</u> , <u>an</u> , <u>en</u>
MOD-ESS <sup>2</sup>	modalis-essive	<u>lag</u> , <u>leg</u>
MUL	multiplicative	<u>szor</u> , <u>szer</u> , <u>ször</u>
NOM	nominative	∅
ORD	ordinal numeral	<u>dik</u> , <u>adik</u> , <u>edik</u> , <u>odik</u> <u>ödik</u>
ORD-MUL	ordinal-multiplicative	<u>adszor</u> , <u>edszer</u> , <u>odszor</u> , <u>ödször</u>
PERS	personal ending	(see Table 3.3.1)
PL	plural	<u>k</u> , <u>ak</u> , <u>ek</u> , <u>ok</u> , <u>ök</u>
PLT	plurale tantum	<u>ék</u>
S	- <u>s</u> (derivative ending)	<u>s</u> , <u>as</u> , <u>es</u> , <u>os</u> , <u>ös</u>
SOC	sociative	<u>stul</u> , <u>stül</u> , <u>astul</u> , <u>estül</u> , <u>ostul</u>
SUB	sublative	<u>ra</u> , <u>re</u>



<u>abbr.</u>	<u>type of the affix</u>	<u>morphemes</u>
SUP	superessive	<u>n</u> , <u>on</u> , <u>en</u> , <u>ön</u>
TEM	temporalis	<u>kor</u>
TER	terminative	<u>ig</u>
U	-ú (derivative ending)	<u>ú</u> , <u>ű</u> , <u>jú</u> , <u>jű</u>
(WQ)	(other forms, see Fig.6.5)	

Table 3.3.1

PERS := PERS SG / PERS PL

PERS SG := PERS-1SG / PERS-2SG / PERS-3SG

PERS-1SG := PERS 1SG / PERS 1SGi

PERS-2SG := PERS 2SG / PERS 2SGi

PERS-3SG := PERS 3SG / PERS 3SGi

PERS PL := PERS-1PL / PERS-2PL / PERS-3PL

PERS-1PL := PERS 1PL / PERS 1PLi

PERS-2PL := PERS 2PL / PERS 2PLi

PERS-3PL := PERS 3PL / PERS 3PLi

PERS 1SG := am / em / om / öm / m

PERS 1SGi := aim / eim / im / jaim / jeim

PERS 2SG := ad / ed / od / öd / d

PERS 2SGi := aid / eid / id / jaid / jeid

PERS 3SG := a / e / ja / je / á / é / já / jé

PERS 3SGi := ai / ei / i / jai / jei

PERS 1PL := unk / ünk / nk

PERS 1PLi := aink / eink / ink / jaink / jeink

PERS 2PL := atok / etek / otok / ötök / tok / tek / tök

PERS 2PLi := aitok / eitek / itek / jaitok / jeitek

PERS 3PL := uk / ük / juk / jük / k

PERS 3PLi := aik / eik / ik / jaik / jeik



#### 4. ELEMENTS OF THE FORMAL MORPHOLOGICAL AND MORPHONOLOGICAL ANALYSIS

Let us now describe some expressions which occur in the description (more exactly: let us describe what they mean in our use) and let us speak about the essence of the program and of the task realized by this program.

The important expressions which we used are the following:

- 4.1 Stem : (1) stem in the sense of the conventional grammar,  
and  
(2) forms of declension which we are not interested in, so we can regard them as stems from the view-point of our analysis.
- 4.2 Affix : elements of a subset of affixes of conventional grammar which we want to separate from the word-form (these are: endings of number, person, possession, of cases; and the word forming suffixes which are considered productive).
- 4.3 Morpheme : stem or affix.
- 4.4 Word-form : concatenation of a stem and  $0, 1, 2, \dots, n$  affixes ( $n$  is bound by the rules of Hungarian morphology).
- 4.5 Sentence : a sequence of word-forms separated by the signs  $\{ , : - \}$  and finished by one of the sentence ending signs  $\{ . ! ? ; \}$ .
- 4.6 Input data required for the morphological analysis :  
an arbitrary sequence of Hungarian word-forms. Since this phase makes no syntactical and semantical parsing, it is indifferent whether the word-forms create "real" sentences.



4.7 Task of the morphological analysis is to decide one by one whether the word-forms of the input sentences are constructed according to the rules of the Hungarian language. That is, we are looking for an answer to the question whether the affixes succeed in an adequate order.

4.8 Task of the morphonological analysis is to establish, whether the adequate allomorph of the recognized affixes is in the word-form and the sequence of morphemes matches the adequate allomorph of the stem.

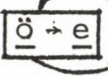
Example 1.

The very morphological analysis will find the word-form \*házakomban (ház+ak+om+ban=house + PL + PERS 1SG + INE [~house+s+my+in]) inaccurate because the -k (sign of the plural) and the possessive affixes can not occur in the same word-form, but it will find the next two word-forms accurate:

házamban } = SUBS + PERS 1SG + INE  
\*házamben }

The analysis taking the phonological rules into consideration, will be performed by the morphonological analyser. It will show also such distinctions as the next example does (Different allomorphs of the same affix stand there, when e.g. the affix PL occurs between the affix and the same stem.):

főnök + höz (SUBS + ADE [~ to the boss])  
főnök + öt (SUBS + ACC [~ the boss + ACC])  
főnök + ök + höz (SUBS + PL + ADE [~ to the bosses])  
főnök + ök + et (SUBS + PL + ACC [~ the bosses + ACC])



4.9 Functioning of the program made by us : all of the word-forms of the given sentences will be analyzed morphologically. If the form is correct, we will produce its decomposition in accordance with all of the possible parsings; if there are no correct forms, we will announce an error.



The error-sign does not consist of the message of the error only, but the supposed affixes will be disjoined in all possible manners(see the next example) and the program constructs the stems obtained in such a manner. The user chooses the correct one of these stems and answers the necessary questions for inclusion into the vocabulary, so + from now on - the stems unknown till now will be known by the program.

Example 2.

Let us assume that our program has to analyze the next sentence:

A csónak a parton feküdt.  
| | | | |  
[the][boat][the][bank] [lie]  
SUP SDPa13

[~ The boat lay on the bank.]

If all the stems of the sentence can be found in the vocabulary, the analysis gives the next result:

DART SUBS DART SUBS+SUP VERB+SDPa13

But if the word csónak[~boat] for example is not an element of the vocabulary, the program will try to define the unknown stem in the following manner:

(1) csónak ~ STEM

\*csóna+k ~ STEM+PL can not be correct, since the vowel at the end of the stem before the affix -k ought to be long.

(2) csón+ak ~ STEM+PL

(3) csó+nak ~ STEM+DAT

(4) csó+nak ~ STEM(verb)+SDPr23

But: when our sentence is



A csónakot vízre tettük.			
[the]	[boat]	[water]	[take]
ACC		SUB	ODPa21

[~ We set the boat on the water.]

the disintegrations of csónakot are the following:

- (1) csónakot ~ STEM  
~~\*csónako+t~~ ~ STEM+ACC can not be correct, because  
the vowel of the stem before the affix -t ought to  
be long
- (2) csónak+ot ~ STEM+ACC

There are no further decompositions, because the apparent plural affix -ak cannot stand before -ot (ACC), only -at; and the affix -nak cannot stand before the ACC because -nak ∈ DAT=CAS, -ot ∈ ACC=CAS and no CAS can occur directly after the other one in the same word-form.

4.10 Final result of the analysis is a set of codes of morphemes defined in the following way:

Let the sentence containing n word-forms have the following form:

$$x_1 | x_2 | \dots | x_n ||$$

where  $x_i$ : an arbitrary word-form,

| : an arbitrary word-form separating sign,

|| : an arbitrary sentence ending sign.

In the first step the analyser E replaces all of the word-forms  $x_i$  with the following form:

$$E^1(x_i) = x_{i_1}^1 + \dots + x_{i_{k_i}}^1$$

where  $x_{i_j}^1$  is the name of the type of the stem or affix

according to the  $j$ -th morpheme of the  $i$ -th word-form containing  $k_i$  morphemes ( $1 \leq i \leq n$ ,  $1 \leq j \leq k_i$ ). The analysis can yield another correct result:

$$E^2(x_i), \dots, E^{m_i}_i(x_i),$$

where  $E^r(x_i)$  is the  $r$ -th succeeded parsing of the morpheme  $x_i$  of our sentence ( $1 \leq r \leq m_i$ , where  $m_i$  is the number of possible correct parsings of the  $i$ -th morpheme).

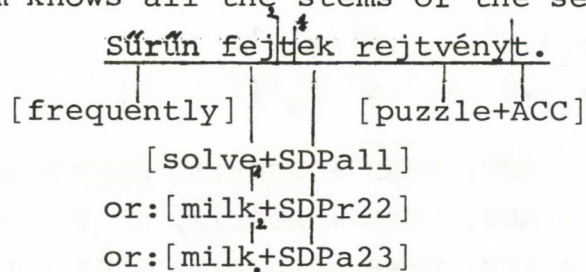
Thus, the set of the sequences of the codes of morphemes which must be transferred to the syntactical analyser, come into being in the following manner:

$$M = \bigtimes_{i=1}^n E(x_i)$$

where  $E(x_i) = \{E^1(x_i), \dots, E^{m_i}(x_i)\}$ .

### Example 3.

If the program knows all the stems of the sentence



(Possible and "impossible" meanings:

- I solve puzzles frequently.
- (\*) You milk puzzles frequently.
- (\*) They milked puzzles frequently.)

The special properties of the word sűrűn:

$$\underline{sűrűn} = \underline{sűrű} + \underline{n} \begin{cases} \swarrow [\text{frequently}] = [\text{frequent} + \text{ADV}] \\ \searrow [\text{dense} + \text{SUP}] \end{cases}$$

Parsing of the sample-sentence:

[n=3]

$$\begin{aligned}
 x_1 : \underline{sűrű} + \underline{n} &\rightarrow \text{ADJ} + \text{ADV} = x_{11}^1 + x_{12}^1 = E^1(x_1) \\
 x_1 : \underline{sűrű} + \underline{n} &\rightarrow \text{ADJ} + \text{SUP} = x_{11}^2 + x_{12}^2 = E^2(x_1) \\
 x_2 : \underline{fej} + \underline{ek} &\rightarrow \text{VERB} + \text{SDPr11} = x_{21}^1 + x_{22}^1 = E^1(x_2) \\
 x_2 : \underline{fej} + \underline{tek} &\rightarrow \text{VERB} + \text{SDPr22} = x_{21}^2 + x_{22}^2 = E^2(x_2)
 \end{aligned}$$



$$x_3 : \underline{\text{fej}} + \underline{\text{tek}} \rightarrow \text{VERB} + \text{SDPa23} = x_{2_1}^3 + x_{2_2}^3 = E^3(x_2)$$

$$x_3 : \underline{\text{rejtvény}} + \underline{\text{t}} \rightarrow \text{SUBS} + \text{ACC} = x_{3_1}^1 + x_{3_2}^1 = E^1(x_3)$$

Thus:

$$M = \bigtimes_{i=1}^n E(x_i), \text{ where } \begin{cases} E(x_1) = \{E^1(x_1), E^2(x_1)\} \\ E(x_2) = \{E^1(x_2), E^2(x_2), E^3(x_2)\} \\ E(x_3) = \{E^1(x_3)\} \end{cases}$$

$$M = \{[E^1(x_1), E^1(x_2), E^1(x_3)], \\ [E^1(x_1), E^2(x_2), E^1(x_3)], \\ [E^1(x_1), E^3(x_2), E^1(x_3)], \\ [E^2(x_1), E^1(x_2), E^1(x_3)], \\ [E^2(x_1), E^2(x_2), E^1(x_3)], \\ [E^2(x_1), E^3(x_2), E^1(x_3)]\} =$$

$$= \{[\text{ADJ} + \text{ADV}, \text{VERB} + \text{SDPr11}, \text{SUBS} + \text{ACC}], \\ [\text{ADJ} + \text{ADV}, \text{VERB} + \text{SDPr22}, \text{SUBS} + \text{ACC}], \\ [\text{ADJ} + \text{ADV}, \text{VERB} + \text{SDPa23}, \text{SUBS} + \text{ACC}], \\ [\text{ADJ} + \text{SUP}, \text{VERB} + \text{SDPr11}, \text{SUBS} + \text{ACC}], \\ [\text{ADJ} + \text{SUP}, \text{VERB} + \text{SDPr22}, \text{SUBS} + \text{ACC}], \\ [\text{ADJ} + \text{SUP}, \text{VERB} + \text{SDPa23}, \text{SUBS} + \text{ACC}]\}.$$

Further on the elements of M will arrive to the entry of the syntactical analyser, i.e. the input data of the next states have arisen as a result of the morphonological analysis.

## 5. LINGUISTIC LAWS ENABLING THE EXAMINATION OF HUNGARIAN WORD-FORMS

### 5.1 Types of nominal stems and ranging of the lexemes made of them

The word "entry" means in our use the first form of a substantive or of a nominal word in the dictionary, which is equal to the nominative in form,



E.g.: LIBA<sub>(entry)</sub> liba<sub>(NOM)</sub>, libá<sub>( (NOM v FOR v TEM) )</sub>.

The entries can be classed according to Papp [8] into groups according to some clear and unstable types of stems. In practice it is worthwhile to distinguish only the entries occurring relatively frequently only from the exceptions which we can enumerate also one by one. In the work of Papp there are 12 clear types. The types in which two different stem-lexemes of the same entry can stand are the unstable types.

Opposite to this stem-classification, which was made for an automatical synthesis, we must produce another division of the set of stems: i.e. classifying by the analytical types which can be used better for automatical analysis.

The essence of it is the following: an analytical type consists of all the stems which can make a word-form only by the same particular set of the types of affixes. Thus, the lexemes of the same entry which can be declined in different ways, will get in different declinational types. (Thus, the analysis does not need the synthetical basis which was indispensable for the synthesis).

## 5.2 Analytical declinational types

If we have a lexeme in the stem-vocabulary, we have to give its declensional type and its vowel-harmony. These two properties give the adequate allomorph which can stand there (after this stem), or they give an answer to whether an allomorph can't occur here at all and vice versa: the conjunction of the concrete affixes gives the declinational type and the vowel-harmony of the stem which they can connect with.

The table of the analytical declinational types can be seen in Table 5.2.1.

The automaton sensible to morphonological rules can be obtained from the automaton making the morphological analysis by dividing the states to more states, according to the number of morphonological realizations.



Table 5.2.1

	NOM	ACC	CAS	PL	PERS 3 SG	Example (low)	Example (high)
1	∅	<u>a/e</u>	∅	<u>a/e</u>	-	<u>ház</u>	<u>térd</u>
2	∅	<u>a/e</u>	∅	<u>a/e</u>	j		<u>vég</u> (anyag)
3	∅	<u>o/ö</u>	∅	<u>o/ö</u>	-	<u>piac</u>	<u>pörc</u>
4	∅	<u>o/ö</u>	∅	<u>o/ö</u>	j	<u>kard</u>	<u>gömb</u>
5	∅	-	∅	<u>a/e</u>	-	<u>báj</u>	<u>rész</u>
6	∅	-	∅	a/e	j	<u>ár</u> (víz)	
7	∅	-	∅	<u>o/ö</u>	-	<u>hús</u>	<u>bőr</u>
8	∅	-	∅	<u>o/ö</u>	j	<u>kar</u>	<u>gör</u> l
9	∅	-	∅	-	j <sup>(1)</sup>	<u>hajó</u>	
10	∅	-	∅	-	j	<u>kocsi</u>	<u>bébi</u>
11	∅	-	∅	-	∅		<u>idő</u>
12	∅	-	∅	∅	∅		<u>ész</u>
13	∅	∅	∅	∅	j	<u>kazal</u>	<u>szemét</u>
14	∅	∅	∅ <sup>(2)</sup>	∅	∅	<u>ló</u>	<u>fű</u>
15	∅	∅	∅	∅	∅	<u>apa</u>	<u>epe</u>
16	∅	-	∅	-	j	<u>almá-</u>	<u>elmé-</u>
17	∅	-	∅	-	∅	<u>apá-</u>	<u>néné-</u>
18	∅	<u>o/ö</u>	∅	<u>o/ö</u>	-	<u>bajsz-</u>	
19	∅	<u>a/e</u>	∅	<u>a/e</u>	-	<u>nyar-</u>	<u>lev-</u>
20	∅	<u>a/e</u>	∅	<u>a/e</u>	∅	<u>ut-</u>	
21	∅	∅	∅	∅	-	<u>fi-</u>	
22	∅	∅	∅	∅	j <sup>(3)</sup>	<u>ap-</u>	<u>ide-</u>
23	∅	∅	∅	∅	j	<u>szá-</u>	

Note: (1) But PERS 3PL without j.

(2) Except SUP.

(3) Only in the PERS 3SG and PERS 3PL, in the other cases ∅.



### 5.3 Morphonological rules

Properly, when we made the declinational types, we spoke about the morphonological rules but in a hidden form.

Now let us examine this question in outlines:

Fitting by the vowel harmony:

stem with low vowel harmony → affix with low vowel harmony  
 stem with high vowel harmony → affix with high vowel harmony  
 stem with mixed vowel harmony → affix with low vowel harmony  
 (usually, but foreign words have special properties).

(i) These rules in themselves are valid when the fitting morpheme has two alternants (a low and a high one). Of course, if there is only one variant, it will connect to the stem independently of the fitting rules.

But in the case of more affixes there exist one-form variants which are able to change the vowel harmony:

$$\begin{array}{ccc} \text{ház} & + \text{PL} & = \text{házak} \\ | & & | \\ \text{[house]} & & \text{[houses]} \end{array}$$

but:

$$\begin{array}{ccccc} \text{ház} & + & \text{beli} & + & \text{PL} & = & \text{házbeli} \\ | & & | & & | & & | \\ \text{[house]} & & & & \text{[ones belonging to the house]} & & \\ & & & & \text{[belonging to]} & & \end{array}$$

(ii) It is worth observing the affixes having one variant only:

é(POSS), éi(POSS PL), ért(CAU), ig(TER), ként(FOR), kor(TEM)

They can stand either after a stem ending in a consonant, or after a stem ending with a or e, or after PERS 3SG CAU and TER can connect to the stem-alternants of the given entry which terminates in á or é; in the case of the other vowels it can be normally concatenated with the stem according to the entry. FOR and TEM can stand after the lexeme according to the entry, or after PERS 3SG's a/e/ja/je, making no change in the stem.



(iii) Affix-morphemes having three kinds of allomorph realization distinguish the high affixes according to labiality/illabiality and not only according to vowel harmony.

Fitting according to labiality (only in the case of high vowel harmony):

- the case of labiality: if the vowel of the last syllable before the affix is ö, ɔ̃, ü or ű, then  
e.g.: ALL = höz.
- the case of illabiality: if the vowel of the last syllable before the affix is e, é, i or í, then  
e.g.: ALL = hez.

The last syllable before the affix is not sure to be the last syllable of the word at the same time, that is another affix can stay between them e.g.: PL or POSS. The possible affixes with ö, ɔ̃, ü, ű are the following:

PL = ök  
 PERS = öm/öd/ötöd/ünk/ük/jük

e.g.: főnök + ök + höz  
           |          |          |  
           [boss] PL ADE [~ to the bosses]

főnök + höz  
           |          |  
           [boss] ADE [~ to the boss]

but: főnök + é + hez  
           |          |          |  
           [boss] POSS ADE [~ to the something of the boss]

főnök + éi + hez  
           |          |          |  
           [boss] POSS PL ADE [~to the things of the boss]

and: főnök + e  
           |          |  
           [boss] PERS 3SG [~ his/her boss]

főnök + é + hez  
           |          |          |  
           [boss] PERS 3SG ADE [to his/her boss]

(iv) Morpheme SUP having four several realizations: three forms behave as we saw in the case when the part of the word-form



without the allomorphs of SUP terminated in a consonant (e.g. STEM, STEM + PL or STEM + PERS). If the stem ended in a vowel and it stands in NOM or after it there stays PERS 3SG, POSS, PL/PERS + POSS after the stem, the adequate form of SUP is n.

- e.g.: (a)  $\frac{f\ddot{o}n\ddot{o}k}{[boss]} + \frac{e}{[PERS\ 3SG]}$  [ $\sim$  his/her boss]  
 $\frac{f\ddot{o}n\ddot{o}k}{[boss]} + \frac{\acute{e}}{[PERS\ 3SG]} + \frac{n}{[SUP]}$  [ $\sim$  on her/his boss]
- (b)  $\frac{f\ddot{o}n\ddot{o}k}{[boss]} + \frac{\acute{e}}{[POSS\ PL]} + \frac{n}{[SUP]}$  [ $\sim$  on the something(s) of the boss]
- (c)  $\frac{f\ddot{o}n\ddot{o}k}{[boss]} + \frac{\ddot{o}k}{[PL]} + \frac{\acute{e}}{[POSS]} + \frac{n}{[SUP]}$  [ $\sim$  on the something(s) of the bosses]

(v) The five several forms of ACC are: at/et/ot/öt/t. The rule system playing here is fairly complicated. Since it is enough for us to receive the occurrence of the given affix and we do not need to offer the motives, we shall survey the morphonological properties of the accusative only in brief.

The variant t can stand in word-forms ending in a vowel (in some words ending in h, j, l, ly, n, ny, r, s, sz, z, zs)

- e.g.:  $\frac{g\ddot{o}z}{[steam]} + \frac{\emptyset}{[NOM]}$        $\frac{g\ddot{o}z}{[steam]} + \frac{t}{[ACC]}$
- but:  $\frac{h\acute{a}z}{[house]} + \frac{\emptyset}{[NOM]}$        $\frac{h\acute{a}z}{[house]} + \frac{at}{[ACC]}$       because it is an ancient affix formed by history

In the case of the word-form ending in a consonant -if the stem occurring in this word-form has a high or mixed vowel -harmony- at or ot stands after it in the case of ACC, as the next part will show:

at can stand there if the stem has only one syllable the vowel of which is a or á and there is no other affix between



the stem and the ACC; e.g.: vár + at (castle + ACC); or: if the stem having no high vowel-harmony and an affix ending in a consonant stays after it (this affix can be only PL, PERS 1SG, PERS2SG, PERS PL), e.g.: kutyá + tok + at (~your dog + ACC); or: it can stand after an arbitrary adjective-noun or numeral-noun having no high vowel-harmony, e.g.: vékony + at (~thin + ACC), nyolc + at (~eight + ACC).

ot can stand there if there is a vowel o, ó, u or ú in the last syllable of the stem and there is no other affix between the stem and the ACC.

et behaves as at but in the case of high vowel-harmony (e, é, i, í).

öt behaves as ot but in the case of high vowel-harmony (e, é, i, í).

Since the rules can be drafted exactly, except for the cases of the stem having already an affix, the vocabulary contains also the declension type of the stem which results the adequate choice of the morphemes to be realized in more than three ways.

(vi) In the following we must deal with the problem of assimilation of the consonants. For the formal handling of full assimilation it is very obvious to induce the following fictive affixes:

$$\text{INS}^2 = *_{\underline{a}\underline{l}} / *_{\underline{e}\underline{l}}$$

$$\text{FAC}^2 = *_{\underline{á}} / *_{\underline{é}}$$

beside the given affixes

$$\text{INS} = \text{W}_{\underline{a}\underline{l}} / \text{W}_{\underline{e}\underline{l}}$$

$$\text{FAC} = \text{W}_{\underline{á}} / \text{W}_{\underline{é}}$$

where:

$$\text{W}_{\text{=}} = \begin{cases} \text{if there is a consonant at the end of the word-form} \\ \text{to be conjugated : this consonant,} \\ \text{if there is nothing, then: } \underline{v}. \end{cases}$$

$$\text{Thus, } \text{INS}^1 = \underline{v}\underline{a}\underline{l} / \underline{v}\underline{e}\underline{l}, \text{ FAC}^1 = \underline{v}\underline{á} / \underline{v}\underline{é}.$$



If the word-form ends in a vowel, according to CAS2 the fitting is the following: after the stem of type 15 and 16, after POSS or PERS 3 SG therestands  $INS^1$  or  $FAC^1$  behind them.

In all other cases if  $INS^2$  or  $FAC^2$  stands at the end of the word-form, the last consonant of the given word-form is double. After its formal control we continue the analysis leaving one of these consonants at the end of the word and we change the morphemes al/el/á/é to the morphemes val/vel/vá/vé.

But now we must reckon with another problem: the Hungarian language does not allow the three identical consonants next to each other. This case can occur in the conjugation of the lexemes ending in a consonant in the cases of the morphemes  $INS$  or  $FAC$ . Thus it seems formal the allomorphs al/el/á/é to come after the stem; e.g. tollal = toll +  $INS$  = toll + val = toll + l + al = \*tollal.

If we can't find the adequate stem among the lexical entries ending with a simple consonant, then we have to, we suppose that our stem ends with a double consonant.

(vii) The allomorph realization of  $PL$  comes into being by the vowels as in the case of  $ACC$ : ak / ek / ok / ök / k. But the rules of fitting become in another manner:

Some remarks connected with this:

k: only after vowels,

ak: usually when the stem (or the last component of the compound) has one syllable and its vowel is a or á, or: the no-entry-lexeme ends in v, e.g. lov + ak (~horses); or: after adjectives which are made by the derivative affix i from nouns e.g. ház + i + ak (~people belonging to the house); or: after original adjective, e.g. okos + ak (~clever ones; but in the form okosok the stem okos is a noun and not an adjective); or: after numerals formed by formative syllables, e.g. hat + szoros + ak (~things which are six times bigger than the other ones).

ok: usually the other cases with low vowel harmony (but



the lexeme kar belongs to this category and this fact shows, the rules are not general. This problem has a historical-phonological reason which we do not employ here.

ek is similar to ak, but in the case of high vowel harmony.

ök is similar to ek, but in the case of high vowel harmony.

(viii) The rules of the realization of PERS are similar to PL in the respects of vowel harmony, in the cases of singular possession, 1st and 2nd person. If the possession is in plural and the stem has a low vowel harmony: aim / aid / aink/aitok; and if the possession is in plural but the stem has a high vowel harmony: eim / eid / eink / eitek.

PERS 3SG and the possession is in singular : a / ja. (low), e / je (high); or a POSS or CAS comes after them their variants: á / já (low), é / jé (high).

PERS 3 SG, but the possession is in plural: if the stem terminates in a vowel (but not i): i ; if the stem terminates in i or usually if the affix PERS 3SG is ja / je : jai / jei; if the stem terminates in a consonant: ai (low or mixed) / ei (high).

PERS 3PL and the possession are in singular: uk / ük, or juk / jük.

PERS 3PL and the possession are in plural: ik, aik / eik, jaik / jeik, similar as in the case of PERS 3 SG.

If the stem terminates in i, the following morphemes can connect to it, in the case of singular possession: m, d, ja / je, nk, tok / tek, juk / jük; in the case of plural possession: jaim / jeim, jaid / jeid, jai / jei, jaink/jeink, jaitok/jeitek, jaik/jeik.



#### 5.4 The Hungarian conjugation from the point of view of morphology

In contradiction to the nominal forms it is not practical to begin to examine the automatical morphological analysis of the Hungarian verbal forms with separating the morphemes following each other, only if the final aim is the full morphological analysis of the verbs' itself.

Our present purpose is a syntactical analysis and our nearest aim is to choose the arrays being relevant to the building of the syntax tree.

#### 5.5 Syntactical relevant verbal affixes

There are five subsets:

- (i) personal suffixes of the verb,
- (ii) personal suffixes of the gerund,
- (iii) derivational affixes of the infinitive,
- (iv) derivational affixes of the adverbial participle,
- (v) the derivational affixes of the verbal noun and the continuous, perfect and future participles.

The verbs derivated by other affixes can be found in the vocabulary as "basic" verbs.

The affixes mentioned above content the next informations:

##### (i) Personal suffixes of the verb:

- (1) type of conjugation: - subjective (indirect)
  - objective with an object in 2nd person (direct<sub>2</sub>)
  - objective with an object in 3rd person (direct<sub>3</sub>)
- (2) mood: - declarative (indicative)
  - conditional
  - subjunctive
- (3) tense: - present
  - past



- (4) number: - singular <1>  
- plural <2>

- (5) person: - 1st  
- 2nd  
- 3rd

(1), (2), (3), (4) and (5) make a complex array.

(ii) Personal suffixes of the gerund:

- (1) number: - singular <1>  
- plural <2>  
(2) person: - 1st  
- 2nd  
- 3rd

(The other informations are in the auxiliary verb of the same given sentence.)

(iii) Derivational affixes of the infinitive:

verbal stem +  $\phi \underline{n_i}$ , where  $\phi = \begin{cases} \phi \\ \underline{a} \\ \underline{e} \end{cases}$

(iv) Derivational affixes of the adverbial participles:

Simultaneous presented mood:

verbal stem +  $\underline{v}A$ , where  $A = \begin{cases} \underline{a} \\ \underline{e} \end{cases}$

Antecedent presented mood:

verbal stem +  $\underline{v}\underline{A}n$ , where  $\underline{A} = \begin{cases} \underline{\acute{a}} \\ \underline{e} \end{cases}$

(v) Derivational affixes of the verbal noun:

verbal stem +  $\underline{\acute{A}}s$ , where  $\underline{\acute{A}} = \begin{cases} \underline{\acute{a}} \\ \underline{e} \end{cases}$

(vi) Derivational affixes of the participles:

Continuous participle:

verbal stem +  $\underline{\acute{O}}$ , where  $\underline{\acute{O}} = \begin{cases} \underline{\acute{o}} \\ \underline{ko} \end{cases}$

Perfect participle:

verbal stem +  $\phi T \underline{t}$ , where  $\phi T = \begin{cases} \phi\phi \\ \phi \underline{t} \\ \underline{at} \\ \underline{et} \\ \underline{ot} \\ \underline{\acute{o}t} \end{cases}$

Future participle:

verbal stem + AndÓ, where  $A..Ó = \begin{cases} \underline{a}..\underline{ó} \\ \underline{e}..\underline{ö} \end{cases}$

## 5.6 The problem of assimilation of consonants in the case of verbs

One of the main problems is the assimilation. In the case of nominal forms we had a usable principle, but now we have also another problems. Therefore we will use - when it is needed - affixes and stems ending in a long consonant. This method cannot be said elegant from a linguistic point of view, but it was impossible to find a formal solution which can be treated easier;

e.g. lássalak = láss + alak  
lát + jalak = verbal stem + O<sub>2</sub>SPr11  
|  
[see]                      [~ that I see you]

## 5.7 Conjugational basis needed at the building of the morphological automaton

As it was already shown, the automaton recognizing the verbal forms separates the stem and the affix at the first step and then it controls, whether the verbal stem belongs to the class determined by the affix-array. We have established to all states of the automaton the set of the stem-types needed to the step to the final state.

We formed the following view-points:

(i) Vowel harmony:

1: <u>o</u>	e.g.: <u>kap</u> + <u>ok</u>
2: <u>e</u>	<u>néz</u> + <u>ek</u>
3: <u>ö</u>	<u>ül</u> + <u>ök</u>



(ii) 2nd person singular:

1: <u>ol</u>	e.g.: <u>bíz</u> + <u>ol</u>
2: <u>el</u>	<u>tesz</u> + <u>el</u>
3: <u>öl</u>	<u>köröz</u> + <u>öl</u>
4: <u>sz</u>	<u>ir</u> + <u>sz</u>
5: <u>asz</u>	<u>ugr</u> + <u>asz</u>
6: <u>esz</u>	<u>költ</u> + <u>esz</u>

(iii) ik in PERS 3SG:

1: <u>ik</u>	e.g.: <u>tör</u> + <u>ik</u>
9: $\emptyset$	<u>lép</u> + $\emptyset$

(iv) Past:

1: <u>a</u>	e.g.: <u>kop</u> + <u>tam</u>
2: <u>e</u>	<u>emel</u> + <u>tem</u>
3: <u>o.a</u>	<u>hall</u> + <u>ottam</u>
4: <u>e.e</u>	<u>sinyl</u> + <u>ettem</u>
5: <u>ö.e</u>	<u>dönt</u> + <u>öttem</u>

(v) Past, 3rd person singular:

1: <u>ott</u>	e.g.: <u>hagy</u> + <u>ott</u>
2: <u>ett</u>	<u>szeret</u> + <u>ett</u>
3: <u>ött</u>	<u>győz</u> + <u>ött</u>
4: <u>tt</u>	<u>lő</u> + <u>tt</u>
5: <u>t</u>	<u>kér</u> + <u>t</u>

(vi) Conditional mood / Infinitive / Gerund:

1: <u>a</u>	-   <u>o</u>	e.g.: <u>vár</u> + <u>na</u> , + $\emptyset$ <u>ni</u> , + <u>nom</u>
2: <u>e</u>	-   <u>e/ö</u>	<u>kérdez</u> + <u>ne</u> , + $\emptyset$ <u>ni</u> , + <u>nem</u> <u>köszön</u> + <u>ne</u> , + $\emptyset$ <u>ni</u> , + <u>nöm</u>
3: <u>a.a</u>	<u>a</u>   <u>a.o</u>	<u>játsz</u> + <u>ana</u> , + <u>ani</u> , + <u>anom</u>
4: <u>e.e</u>	<u>e</u>   <u>e.e</u>	<u>kérd</u> + <u>ene</u> , + <u>eni</u> , + <u>enem</u>
5: <u>nne</u>	<u>nn</u>   <u>nne</u>	<u>te</u> + <u>nne</u> , + <u>nni</u> , + <u>nnem</u>

(vii) Categories (table of affixes which can be connected with the given stem)



### 5.8 Algorithm of the examination of verbal forms

With the help of the tables (and on the basis of the first six viewpoints) we can make the "feature-sixtuple" of the verb wanted to be examined by writing the adequate numbers after each other.

Functioning of the algorithm: if the distinction of an affix from the end of the word-form is succeeded, the affix will define one of the feature-sixtuples mentioned above and it will be tried to match with the feature-sixtuples of the verbs. If it succeeds, we must revise whether the affix connects to the adequate form of the verbal stem.

### 6. DESCRIPTION OF THE AUTOMATON PERFORMING THE MORPHOLOGICAL ANALYSIS

The setting up of the automaton being used to realize the analysis can be followed closely in more steps.

It is obvious that the automaton which analyzes the nominal forms will be more complicated than the automaton analyzing the verbal forms, i.e. we can separate the verbal affixes as complex arrays from the stem. Consequently we can consider all verbal forms as a connection of a stem and an affix (this affix can be also  $\emptyset$ -morpheme, e.g. the SDPr13). For this reason we can store the form of all verbal affix-arrays and the 5-tuple of information belonging to these affixes (type of conjugation, tense, mood, number, person) in the vocabulary of affixes.

The participles come into being by the affixes connected to the verbal stems, thus their parsing begins in this automaton, but it does not terminate there after the examination of certain participles, (for example: the continuous participles will behave as substantives from the point of view of the possible affixes which can be obtained by these participles). We shall come back to the solution of this problem later.



We begin the examination of nominal forms with the examination of the substantives. (Nominal forms: forms of substantives, adjectives, pronouns, numerals.) The schematic diagram of a nominal form is shown by the Fig.6.1.

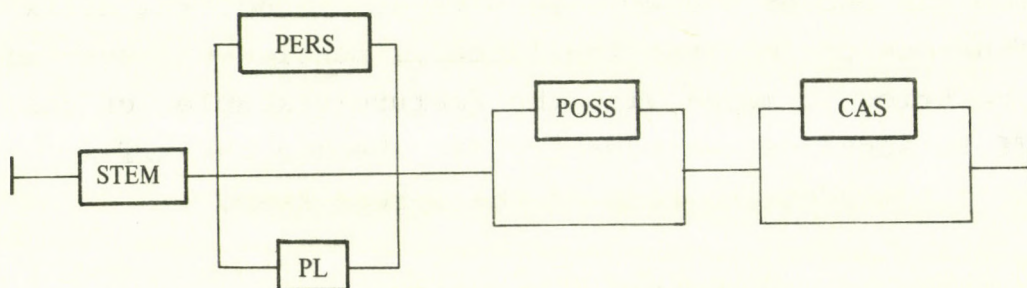


Fig. 6.1

Comment of Fig.6.1. and next similar figures:

Let us try to pass from the left edge (┌) along the lines to the right terminal sign (┐). The direction of the progression can be up, down and right; the only stipulation is :no progressing to left. An arbitrary representative of the affix-types being written in the rectangles having touched along the way can appear on the given place. So we can produce all of the Hungarian nominal forms which can be made of the constituents occurring in the figure by the all possible ranging over of the diagram. It is obvious that we can make further enlargements because the figure does not contain word-forming affixes and other special affixes. We change the box denoted with STEM to a more complicated one. We can build in, not only the affixes not classified up to now but the special adjectival, numeral and postpositional affixes, even the entries coming from the verb.

Thus, instead of the verb's form we have made (Fig.6.2.)

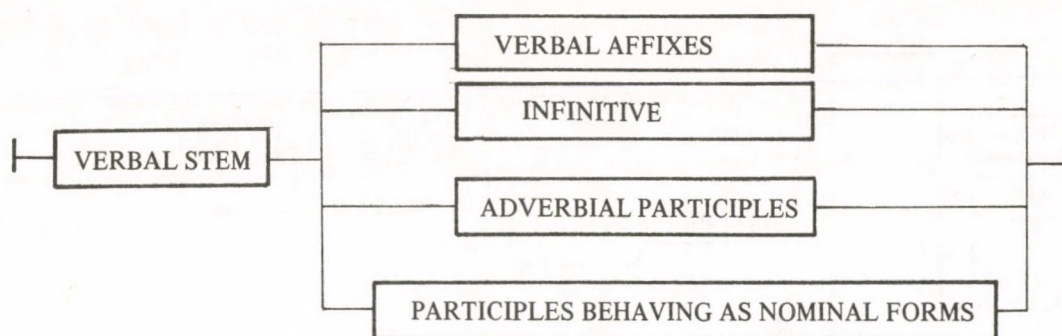


Fig. 6.2

we shall get the next modified form (Fig.6.3.).

And instead of the present form of the substantives (Fig.6.1) it was worth developing the form of Fig.6.4.

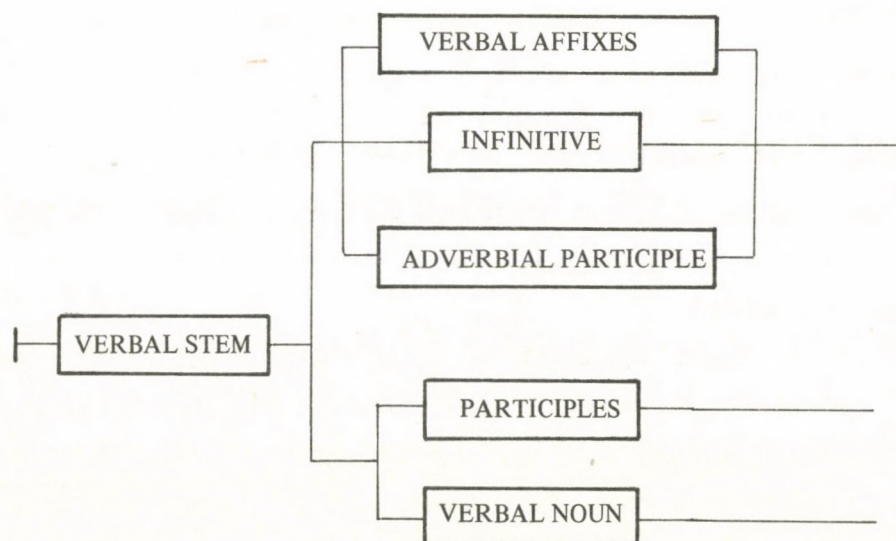


Fig. 6.3



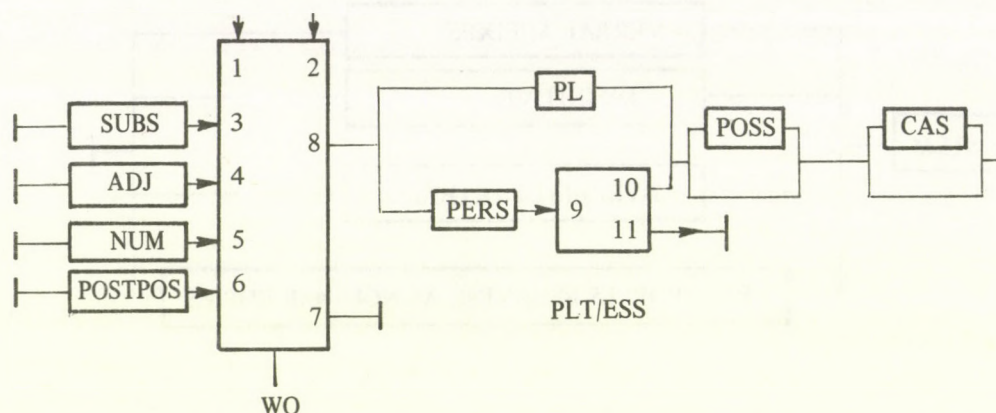


Fig. 6.4

One of the new units appearing after the stem obtained collective designation WQ, the other got the name PLT/ESS because the affixes plurale tantum and essivemodalis will be analysed here.

Of course these affixes of nouns, adjectives, articles etc. are the affixes only we thought practical to distinguish on the momentary level of parsing.

If we pass along on the figures before, from right to left we get the building of the automaton performing the morphological analysis.

Naturally the morphonological analysis comes next yet by working out the interior of such a rectangle. Its basic principle the affix recognized already, determines what kinds of phonological properties the preceding affixes can have, and the program culls from the set of affixes which is restricted in this manner when the program enters a given "box".

Thus, the diagram of the automaton performing the morphological analysis (Fig. 6.7.).

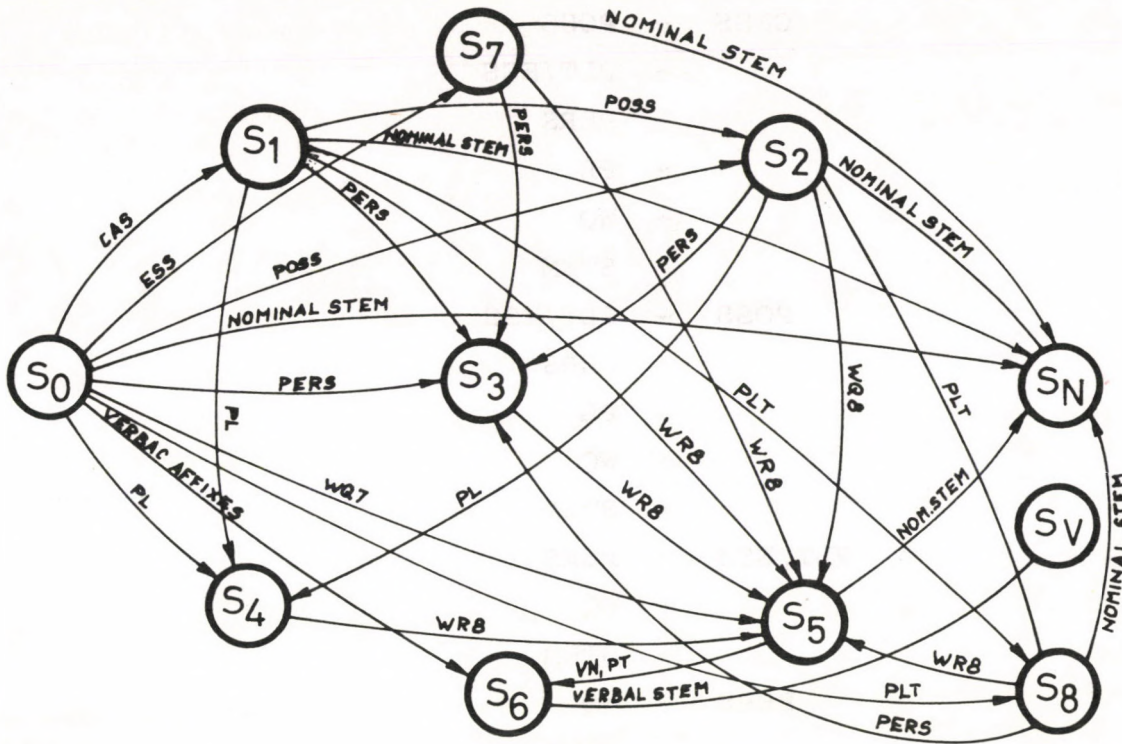


Fig. 6.7

Note:

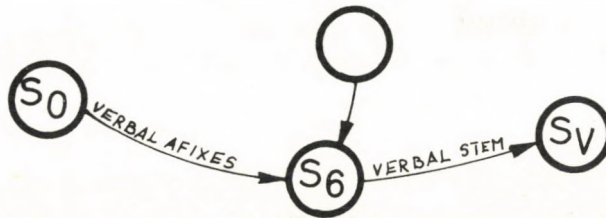


Fig. 6.8

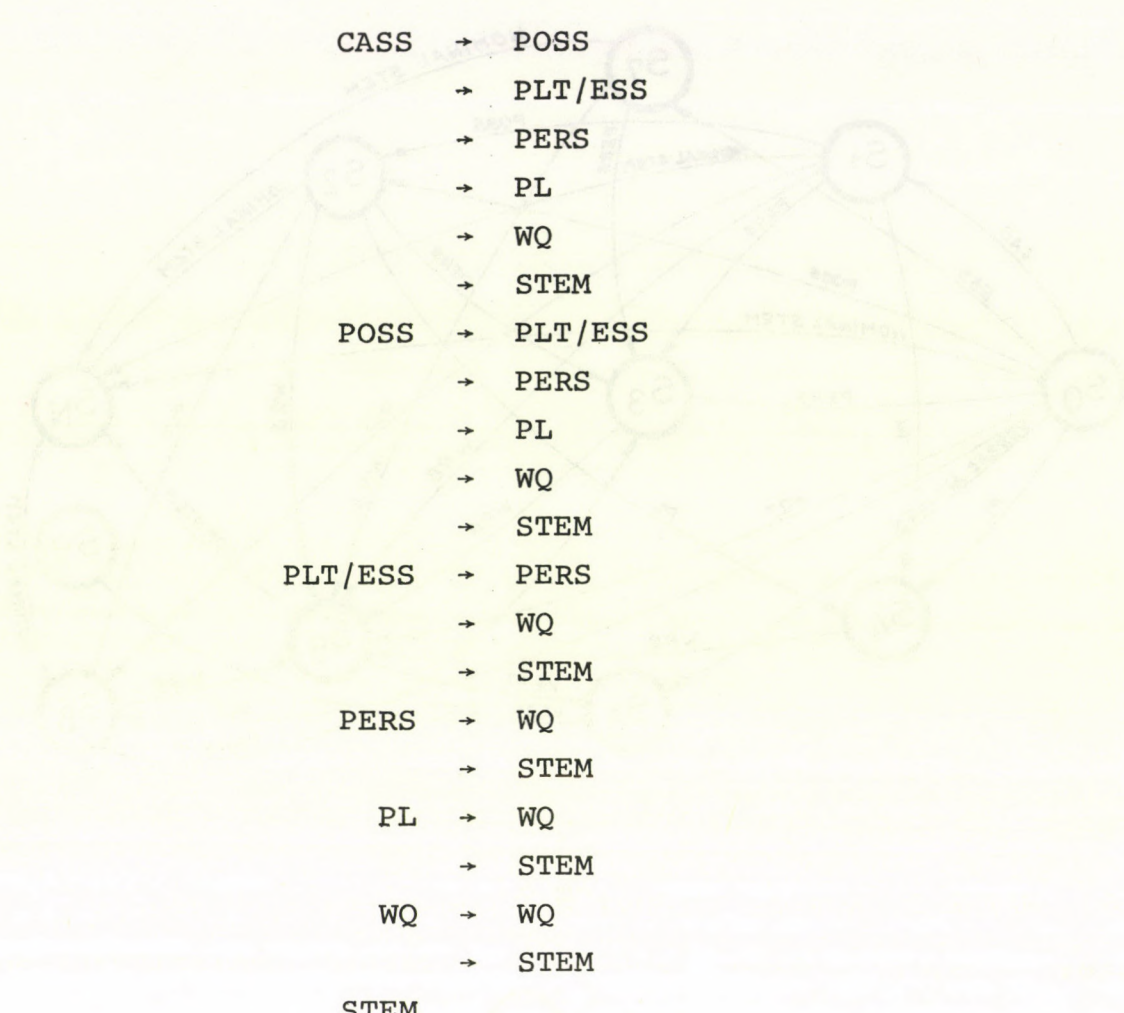
That is, the verbal automaton treated independently up to now, is built into the nominal automaton.

The algorithm performing the morphological analysis is shown by the next table.

The morpheme on the left of  $\longrightarrow$  can be preceded by the morphemes standing on the right of  $\longrightarrow$  only.

The classification of the table is suitable for Fig. 6.4., that is, the next combinations are possible:





CASS	→	POSS
	→	PLT/ESS
	→	PERS
	→	PL
	→	WQ
	→	STEM
POSS	→	PLT/ESS
	→	PERS
	→	PL
	→	WQ
	→	STEM
PLT/ESS	→	PERS
	→	WQ
	→	STEM
PERS	→	WQ
	→	STEM
PL	→	WQ
	→	STEM
WQ	→	WQ
	→	STEM
STEM		

## 7. CONCLUSION

We realized the automaton performing the morphological and morphonological analysis of Hungarian word-forms in PROLOG. The solution principle of the problem did not contain details not indifferent from the point of view of the practical realization. Such detail is necessary in the organization of a great vocabulary of stems in the making of the tree-form. If this is not done, the program can be very slow. The acceleration of the algorithm, however, is possible in more ways, but our fundamental task was to try out a method which is able to solve this task and to verify the adequate functioning of this method.

The input data of our next program - the syntactical analyzer - will be the output data of the morphological analyzer which we have been just made.



## 8. BIBLIOGRAPHY

- [1] ANTAL L. A magyar esetrendszer (The Hungarian Case System). Akadémiai, Bp. 1961.
- [2] ANTAL L. Egy új magyar nyelvtan felé (Towards a New Hungarian Grammar). Magvető, Bp. 1978.
- [3] HELL GY. Mechanical Analysis of Hungarian Word Forms. Computational Linguistics & Computer Languages X. p.125-134.
- [4] JÁNOSKA S. A magyar ige automatikus toldalékolásának egy modellje (A Model of the Automatical Affixation of Hungarian verbs ). Nyelvtud. Közl. 58. Akadémiai, Bp. 1967.
- [5] LUGOSI PAP M. One Model of the Hungarian Verb Synthesis Computational Linguistics and Computer Languages X. p.39-98.
- [6] MÁTHÉ J, KOVÁCS BÖLÖNYI E., SCHWEIGER P., SZÉKELY E. A magyar igeragozás független analízisének egy modelljéről (About a Model of the Independent Analysis of Hungarian Conjugation). Nyelvtud. Közl. 58. Akadémiai, Bp. 1967.
- [7] MELCSUK, I. A magyar főnévragozás egy modellje (A Model of Hungarian Declination). Nyelvtud. Közl. 58. Akadémiai, Bp. 1967.
- [8] PAPP F. A magyar főnév paradigmatis rendszere (The Paradigmatic System of the Hungarian Substantive). Akadémiai, Bp. 1975.
- [9] PAPP F. Machines in the Service of the Hungarian Substantive as a Machine. Computational Linguistics and Computer Languages XIII.
- [10] PRÓSZÉKY G., TÓTH L. Magyar nyelvű mondatok számítógépes szintaktikai vizsgálata (A Computational Syntactical Analysis of Hungarian Sentences). ELTE Dissertation, 1979.



- [11] STEIN M. A magyar főnév szintézise elektronikus számítógéppel (The Synthesis of the Hungarian Substantive by Computer). Nyelvtud.Közl.58. Akadémiai, Bp. 1967.
- [12] Tanulmányok a magyar nyelv szófajtana és alaktana köréből (Studies of the Lexicology and Morphology of the Hungarian Language). (Ed. by Rácz E., Szatmári I.). Tankönyvkiadó, 1974.

## LANGUAGE DESIGN OBJECTIVES AND THE CHANGE SYSTEM

*Endre Simon*

Research Group on the Theory of Automata  
Szeged, Hungary

### Abstract

It is the intent of this paper to describe the main characteristics of programming language design, and to give a short presentation of the CHANGE system. The CHANGE programming system was developed for adequate solution of various kinds of problems by multiprocessing and language extension. The paper consists of two parts to emphasize some existing language aspects and to characterize the CHANGE system in comparison with them.

### 1. Introduction

The 30 years development of programming languages may be characterized by three phases corresponding roughly to the discovery and description of programming language concepts, the elaboration and analysis of concepts developed earlier, and the further development of the 70's influenced by the development of an effective software technology. This discussion of programming language development is far of course from complete. There are both practical developments such as special purpose languages and theoretical topics. Computer programming today is in the state of creative ferment. There is a growing recognition that the available programming languages are not stimulating for designing *computer* and multiprocessor *systems*. When we look at the integrated multiprocessor systems that will soon dominate computing, the situation is even worse. In order to develop such systems, we need to shift our attention away from the detailed specification of algorithms, towards the descrip-



tion of the properties of packages and objects with which we build.

There is another conception dividing the history of computer languages into four periods. The first period was that of *machine languages* with uncomfortable coding conventions for directing the machine through a long sequence of elementary actions. The second period was the time of *symbolic machine (assembly) languages*, which made the algorithm's coding task simpler, but not less machine-dependent. The third period was the time of *high-level languages*, that added a significant level of economy in writing, machine independence, and readability of programs. The fourth period appears to be that of *very high-level languages*, which enable the writer to express "what" is to be done, in a language more familiar to his discipline, rather than "how" it is to be done.

One of the more interesting facts that emerge from a study of programming language development is the remarkable stability of early programming languages like FORTRAN, COBOL and so on. These languages have satisfied the claims of a wide-ranging class of users to solve their problems. But programmers should never be satisfied with languages which permit them to program everything. There is a need for languages which help the programmer to find the most natural representation for the data structures he uses and operations he wants to perform on them is clearly illustrated by the current trends in the evolution of programming languages.

The programming language CHANGE ([Leg 73], [Leg 80]) as a universal base language is not qualified to satisfy every criterion motivated by the users, but takes the evolution of languages into consideration. The main purpose of CHANGE is to give an adequate possibility to the user to specify his problem on different levels of abstraction by *dynamic language extensibility*. Furthermore there is another language feature which assures multiprocessing orientation and maximal access to the CHANGE interpreter on different levels of CHANGE programs. In other words we can consider each CHANGE program as a multiprocessor which has a finite number of active and passive



processors in every instant. During the execution of a CHANGE program there exists a language tool to generate new processors which have private data. Processors can build a processor structure for the dynamic description of control structure. A very important language feature is that the user is able to *operate on the CHANGE statements* during program execution.

In the next paragraph of this paper we attempt to give a brief summary on the design of programming languages. We would like to make it perfectly clear that we refer to language features which have connections to the peculiarities of the programming system CHANGE.

## 2. Standpoints to language design

Designing a high-level programming language is at best a difficult task. The designer must keep the entire design in mind at the top level in order to decide what the various components of the language should be, what they should look like, and what effect they have one upon the other. And if he would settle on a design, and then change some aspect of it, he must be concerned with the effect this change would have on the rest of the language. These problems are made more difficult if any of the features of the language is new or has never been used in combination with the others [Bas 75].

It is easy to see that there exists a set of semantic features which are common to many languages. These include such elements as assignment, block structure, functions and procedures, transfer of control, parameter passing, data structures and input/output which can be used as an aid to the understanding of languages in general. Nevertheless, programming language definitions are intended to serve at least the following two purposes. As a specification of "correctness" for the language implementer, and as information for the user who wishes to determine whether a program performs its intended task. The formal definition of the CHANGE language based on VDL (Vienna Definition Language) will be described in a following paper.



The design of a programming language is based on the selection of the basic set of primitives which define the *problem area* addressed by the language. These primitives form the basic data types and operations of the language. Once discovered and chosen, they are embedded in a set of control and data structures appropriate for expressing algorithms in the problem area. The role of this phase is to specify informally these basic language concepts and to define a full component of *language* features necessary to develop a viable programming language design.

The second phase of the language design consists of formalizing the discussion stage by representing the design in an *interpretive semantic model*. Ideally, the design should be representable on many levels so that the designer can get the whole picture at whatever level of detail is needed and minor changes in the design can be represented by minor changes in the model at the right level. The problem in this phase is that a complete interpretive semantic model usually requires too much details to be of much information at this level. The solution proposed in [Bas 75] is to initially model the language at a very high level using a model which allows a hierarchical and modular specification of the design. Then in *successive passes* a remodelling of different aspects of the design can take place, specifying the structures at lower and lower levels. This is similar to the approach taken in writing a structured program. This refining process involves the introduction of more structuring into the features of the language [Tuc 75].

It should be noted that the CHANGE language which has features such as operation (to be executed) on statements was implemented in a natural way on the basis of this phase (c.f. Appendix F in [Leg 80]).

Phase 3 of the design process is the design of the implementation of the language. This has been the traditional use of interpretive semantic models. Implementation of a programming language can be done by writing a *compiler* or *parser-interpreter* system. In the second case, the parser transforms a source



text into an intermediate code which is a data structure to be selected on the base of effective interpretation. The model for the compiler/parser design in phase 3 follows naturally from the model of the language design in phase 2 continuing the refining process towards the level of implementation. A formal model for the design of the compiler enhances the clarity of the design and provides a vehicle for proving the correctness of the compiler.

In the following we give some basic language features in order to make a comparison between existing programming languages and the CHANGE system.

### 2.1. Some language features from the point of view of the CHANGE

As P. Wegner said, we have learned that in order to achieve flexibility and power of expression in programming languages we must pay the price of greater complexity [Weg 76]. In the 70's there was a tendency to retrench towards simpler languages like Concurrent Pascal, even at the price of restricted flexibility and power of expression. At the same time there was another tendency which emphasized *extensible languages*, as an alternative approach to the design of general-purpose languages. An extensible language starts off with only a few features, but it can be extended by the user who can define additional features as he needs them.

The objective aimed at in the design of an extensible language is the provision of a variable and flexible means for the description of data structures and operations on them, together with the provision of a sufficiently flexible syntax and semantics so that algorithms could be expressed in natural manner. In general, a program in an extensible language will consist of a combination of a base text (in the base language) and an *augment text* expressed in terms of constructs which are not part of the base language but which have been suitably defined. The vehicle for the introduction of new constructs into the base language is provided by the implementation of a *definition language*, statements in which the definitions of



constructs which are in an *augment language*, are formed. The combination of the augment text and the base text forms a text in an *extended language*. If the definition language is a subset of the base language then the base language may be called as a *self-extending language* [Sol 71]. In any case, the definition language and the corresponding extension mechanism should be powerful enough for the introduction of

- (i) new data types
- (ii) new data transformations
- (iii) new modes of sequencing of both the primitive and defined data transformations
- (iv) new statement types or new syntactic forms.

We introduce the term *derived language* to denote the language into which the statements of the augment text are converted by the extension mechanism. The distinction between the derived and extended languages and texts is important because the translation from source text to target text in general consists of several passes of compilation.

From the implementation's point of view of an extensible programming language we can distinguish compiler and parser-interpreter systems too. In the first case the compiler of such a language have built-in *open-endedness* which allows the user to tailor the language to his specific needs. Then, the extension mechanism which is a part of the compiler is called *static extension mechanism*. If the extension mechanism is a part of the language interpreter we call it *dynamic extension mechanism*.

The classification of static extension mechanisms can thus be made on the basis of the stage of the translation process during which the definition of an extension is processed and the augment text is converted into a text in the derived language which is then merged with the translation of the base text into the derived language. In this case an advantage of an extensible language is that language features which are never used at a given installation never need to be implemented, so that hopefully the size of a compiler can be kept under control.



In the case of dynamic extension, statements are converted into the *intermediate text* in a unified manner. Only syntactical properties of the augment statements are checked by the compiler on the base of definitions written in the definition language. As we are going to see in the next chapter the CHANGE system is a typical example for dynamic extension. The dynamic extension gives an effective means for the user to make a flexible implementation of various kinds of problems. In the CHANGE system the *extension of data types is static*, i.e. it is performed at compile time.

In the following we give a short presentation of a system called ECL [Wgb 71] developed at Harvard University which has some features which are similar to the main characteristics of the CHANGE system. The most important goals of the ECL system are:

- (i) to allow problem oriented description of algorithms, data and control over wide range of application areas
- (ii) to facilitate program construction and debugging
- (iii) to allow and assist in the development of highly efficient programs
- (iv) to facilitate smooth progression between initial program construction and the final realization of an efficient software product.

To aid program construction and debugging, the ECL system has been designed for use in an interactive, on-line fashion.

The ECL system reads and parses each command, interpretes it and turns to the next command. Since commands include calls on procedures which may be programmer-defined, the interpretation portion of the cycle may set of the running of a compiled program. At the heart of the ECL is the command handler, the routine which controls the above command loop. It has two main components: the *parser* and the *interpreter*. The parser calls on a *lexical analyzer* to decompose the input stream into *tokens*. Both the input source and the parse tables may be changed by commands, so that the source of commands and the *language* in



which commands are expressed are subject to *change by programmer*. This mechanism may be called *static extension*. The output of the parser is a representation of the commands as linked lists called *intermediary lists*. The list structured representation has two uses. On the one hand, it can be executed directly by the interpreter, on the other, it is a convenient form of input to the compiler. This achieves several economies. A *program* needs to be *parsed only once*, on input. Hence the interpreter does not reparse a line each time it is encountered during execution, e.g., in a loop. Also, the compiler is considerably simplified since it is not at all concerned with parsing.

Most commands will be *function calls*, i.e., the application of a routine, for example:

- (i) a routine for defining new procedures and operators
- (ii) the compiler
- (iii) routines to define new data types
- (iv) routines to change the parse tables, thereby changing the syntax of the language.

It should be clear that ECL is an eclectic system, which provides definition mechanisms for extension along three axes: *syntax*, *data types* and *control structures*. The idea is to construct a small initial system consisting mostly of powerful definition mechanism for *self-extension*. Only the initial system needs to be implemented and maintained by the system creators.

In order to give a short comparison between ECL and the CHANGE systems from the point of view of language extension, too, let us examine now the ECL system. The language may be extended by

- (i) adding to the syntax specification *new syntax rules with augments*
- (ii) defining the function names used as prefix operators in the new intermediary list form, thereby defining the *semantic specification*



- (iii) adding the *parse table generator* based on the new syntax specification
- (iv) *switching the parser* to be driven by the resulting new parse tables.

Compiling the program and the semantic specification functions will yield an acceptable although not specially optimized code for the new construct.

The main differences between ECL and the CHANGE system are the following.

- (i) The ECL system has been designed for use in an *interactive*, on-line way, not neglecting batch processing [Wgb 71], while the CHANGE system can be used only in *batch* manner.
- (ii) In the ECL system, during interpretation there is no extension, namely the extension mechanism is a so called *static extension*. In the CHANGE system those extensions which refer to *data types* are completed by the parser in a *static* manner. The *definitions* of statement extensions and the *augment statements* are processed by the CHANGE interpreter. Hence, the extension mechanism, which is a part of the interpreter is called *dynamic*.
- (iii) From some points of view the ECL system is considered more intelligent as the CHANGE system, but it should be noted that the size of ECL is about 20 times greater as the size of the CHANGE system.

There are two essential aspects of forming view of the two systems on the basis of extension mechanisms. From the user's point of view the *extension mechanism of the CHANGE system is more efficient* because the user can control statement extension under the execution of it's program to introduce various kinds of syntactic and semantic constructs. The cost of this powerful tool is that the efficiency of the implementation will be decreased by the execution speed of the interpreter. In our opinion, separating data type and statement (or



control) extension in the CHANGE system gives a sensible compromise from both points of view.

When an extensible language will be implemented by a compiler it can be implemented by an *extensible compiler-compiler system*, too [Nap 80]. There is another example of the implementation of extensible languages by a compiler-compiler system. The Helsinki Language Processor (called HLP) is a compiler-compiler system which has been developed at the University of Helsinki [Rai 78]. The system consists of modules for the *lexical*, *syntactic* and *semantic* processing of computer languages. Semantic processing is carried out on the basis of *attribute grammars*. The lexical and syntactic metalanguages of the original system were increased by us so that the user could define the compiler of a self-extending language, too. In our system the *derived language* may be the *base language* and an *attributed tree language*, too [Sim 81]. Our implementation - the first one in Hungary - will be completed at the end of this year. After that the parser of the CHANGE system which is now running on the CDC 3300 computer will be rewritten in the HLP system.

In our opinion the CHANGE system has another basic feature too which is called *multiprocessor-orientation* [Leg 73]. In order to give some new comparative points of view in connection with the CHANGE system we introduce some high-level language constructs on the basis of Concurrent Pascal which are used to describe *concurrent processes*. A process is a sequence of "operations" carried out one at a time. Processes are concurrent if their executions *overlap* in time. The language notation

$$cobegin S_1; \dots; S_n coend$$

is treated by the compiler so that the statements  $S_1, \dots, S_n$  can be executed concurrently. Concurrent statements can be arbitrarily *nested* thus determining a *process hierarchy*. It is this hierarchy which determines the *processor hierarchy*.

In the CHANGE system it was developed by another concept. The processor hierarchy is programmable, that is it can dynamically be formed during program execution by the statements



COPROCESSOR and SUBPROCESSOR. Every CHANGE processor has a *private library* to define the semantics of the statements. In any step the next statement of the *active processors* is carried out by the CHANGE interpreter. The augment statements typed CLOSE EXTEND are processed in one step too.

In high-level languages for concurrent processes which have access to common data there are some *synchronizing tools* such as conditional critical region, semaphore, monitor and event queues. Because the CHANGE language has *common data* with the except of closed extend's data, there is a need to introduce some kind of synchronizing primitive. For the CHANGE processors the language contains statements called WAIT and NOWAIT to control the working of those. It should be noted that the *semaphore* as a synchronizing primitive is able to solve all synchronizing problems connected with concurrent processes. Variables of the *type semaphore* can be introduced to the CHANGE language by *data type extension*. The *semaphore operations* called P and V by Dijkstra are considered as the results of *statement extension*. The classical Readers and Writers problem solved with semaphores can be found in the next chapter.

### 3. The CHANGE programming system

As we have seen the CHANGE programming system consists of two main components. The CHANGE *parser* maps the source text into an intermediate text after carrying out the static extension and the syntax analysis of the base and augment statements. The intermediate text will be executed by the CHANGE *interpreter*. During interpretation the dynamic extension of intermediate augment statements is executed and the processor architecture is formed. In the present chapter the basic concept of the CHANGE system will be presented first. A short presentation of the CHANGE statements will follow. Finally some typical applications are given in order to demonstrate the usefulness of the CHANGE language.



### 3.1. The concept of the CHANGE system

The hardware and software conditions which led to the development of the CHANGE are outlined by the designer of the language [Leg 73], [Leg 80]. As it appears to the author the problem area for which the use of the system is adequate depends on the *choosing of the base language*. The basic data types and statements of the CHANGE are *FORTTRAN-like*. Statements for *extension, processor management* and for *program modification* have FORTRAN-like syntactic properties too. Because the CHANGE system was developed as a parser-interpreter system, the FORTRAN-like declarations are treated as statements. It is easy to see that the semantics of the CHANGE statements has many *dynamic properties* to inspire the user writing effective but unstructured programs.

In our opinion the development of the language has the following three motivations:

- (i) It must give some kind of possibility to the users to introduce some *new language constructs* to specify their problems at different levels of abstraction, supposing that enough primitive operations can be expressed on the base language.
- (ii) The base language must contain tools for creating *concurrent processes* and expressing different types of synchronization between them. Processors assigned to the processes form a processor hierarchy which can be reconfigured during the execution of a program. Every processor can assign various kinds of semantics to the statements.
- (iii) It should assure the language *maximal access* to the state's components of the interpreter.

The main power of the CHANGE system - distinguished from other systems - is the facility of the *open and closed statement* extensions. In the case of open extension after the parameter passing the augment statement is replaced by the body as an open routine which can contain some other augment state-



ments. The closed augment statement is interpreted as a closed routine. After parameter passing the *statement body* is carried out as a single statement in a new processor level with private data.

The introduction of the motivation (iii) is considered unnecessary, for it gives possibility to the user to develop so called *unstructured CHANGE programs*. Therefore the CHANGE system has different levels of the *trace* facilities for debugging.

There is a special statement called *CHANGE statement* which can modify several parts of the program during it's execution. In case of application of the CHANGE statement in a *statementbody* the forming of the semantics of an augment statement can be controlled by the actual parameters. Let us assume that in a state of the interpretation of a CHANGE program some open augment and CHANGE statement is executed. Consequently, the original program has been modified. We can reproduce now several parts of the program on FORTRAN and CHANGE level too, by the execution of the TRANSLATE and LIST statements.

An interesting programming method using the extension facility, the TRANSLATE and LIST statements will be given next. If we wish to solve some computational task by the method of *step by step refinement* of the *abstraction levels* then we can do the following. At the first abstraction level we formulate our task by forming extended data types and control structure. At the subsequent levels we give the definitions of augments until the statements of the base language are reached. At the optional level of the implementation the modified CHANGE program can be reproduced by using LIST statement to control the generated program. If the program generated at the last level does not contain some special statements such as processor management statements etc. then the user can apply the statement TRANSLATE which generates the equivalent debugged FORTRAN program in order to optimize run time.



### 3.2. On the statements of the language

In order to give a short presentation on the statements of the language let us consider as an example the following program. The classical *Reader and Writer problem* is solved by a *cyclic message buffer*.

The program consists of two parts (see Program 1. and Program 2.). The first part is to specify

- the extension of an integer basic data type in a static manner to introduce a new data type called SEMAPHORE (see line labeled 20)
- the declaration and assignement of the initial value for the text variables O and P (see lines labeled 21-23)
- the open statement extension (in the line labeled 24) which is introduced for the specification of the initial assignement to variables typed SEMAPHORE
- the open statement extensions in order to introduce the semaphore operations P and V (see lines labeled 25-26)
- the open statement extension for incrementing the pointers of the message buffer (see line 27)

At compile time after the processing of extensions the syntax of the augments is transferred to the *syntax library* (identified 49). During the execution of Program 1. the semantics of augments is copied into the *semantics library* (identified 50) and the files which contain the libraries are rewound (see line labeled 28-30). The specification of the *parameter passing* can be seen, for example, in the line labeled 4, 9, and 13. The description of a statement body can be found in the lines 2-3.

C THE READER AND WRITER PROBLEM

C SOLVING WITH SEMAPHORES

```
20  EXTEND TYPE SEMAPHORE,INT
21  TEXT O,P
22  $OPEN$ TO O
23  $PERM$ TO P
24  EXTEND O,P,SEMANTICS AT 7, $SEMAPHORE:V=INT:E$
25  EXTEND O,P,SEMANTICS AT 1, $P(SEMAPHORE:V)$
26  EXTEND O,P,SEMANTICS AT 5, $V(SEMAPHORE:V)$
```



```
27    EXTEND O,P,SEMANTICS AT 10, $INCREMENT INT:V MOD INT:E$
28    LIBRARY TO FILE 50
29    REWIND 49
30    REWIND 50
      STOP
1     SEMANTICS BODY 2-3, PARAMETERS 4-4
4     I==SEMAPHORE (1),INT(1)
9     J==INT(1)
13    K==INT(2)
2     IF(I.LE.O) 2,3
3     I=I-I
5     SEMANTICS BODY 6-6, PARAMETERS 4-4
6     I=I+I
7     SEMANTICS BODY 8-8, PARAMETERS 4-9
8     I=J
10    SEMANTICS BODY 11-12, PARAMETERS 9-13
11    IF(J.LT.K) 12,13
14    J=O
12    J=J+1
```

Program 1.

In the Program 2. four variables are declared as extended type called SEMAPHORE on the basis of the extended syntax library. After the initialization of the pointers and semaphore variables two processors are initiated and a P operation is executed to await the termination of two subprocessors. The first processor labeled 2 is treated as the *Writer* and the second labeled 3 as the *Reader*. The access of the message buffer is synchronized by semaphore S1. The read and write operations are synchronized using semaphore operations on the variables S3 and S2.

```
LIBRARY FROM FILE 50
INT B,C,BUFFER(10),I,J,K,L
SEMAPHORE SO,S1,S2,S3
B=1
C=1
SO=-1
```



```
S1=1
S2=10
S3=0
SUBPROCESSOR 2,2
SUBPROCESSOR 3,3
P(SO)
STOP

C PROGRAM FOR PROCESSOR 2
2   LIBRARY FROM PROCESSOR 1
    DO 4 I=1,30,1
    READ(60,$(I4)$) K
    P(S2)
    P(S1)
    BUFFER(B)=K
    INCREMENT B MOD 10
    V(S1)
    V(S3)
4   REPEAT
    V(SO)
    STOP

C PROGRAM FOR PROCESSOR 3
3   LIBRARY FROM PROCESSOR 1
    DO 5 J=1,30,1
    P(S3)
    P(S1)
    L=BUFFER(C)
    INCREMENT C MOD 10
    V(S1)
    V(S2)
    WRITE(61,$(1X,I4)$)L
5   REPEAT
    V(SO)
    STOP
```

Program 2.



### 3.3. Application examples

The CHANGE programming system is now running on some computers such as CDC 3300, R-40 and IBM 3031. In all cases the language of the implementation is the FORTRAN language. At present about 20 000 lines of CHANGE programs are tested and integrated in a source library. In the present chapter some typical applications of the CHANGE will be briefly presented on the basis of the CHANGE library.

The first example is an application of the CHANGE processor hierarchy to sort various types of data elements referred to [Tho 77]. As the second example a system for education is presented. On the first level the system called CAI\* (Computer Aided Instruction) defines a large number of elementary exercises as augment statements. The statement body of augments determines the algorithm to solve the exercise. Using multi-level extension complex exercises can be composed from the elementary ones. Extension can be used to give different formulation to exercises with identical solutions described by a statement body.

There exists an application of CHANGE to produce programs for NC (NUMERICAL CONTROL) machines. As a first step basic operations of the machine are defined as augment statements. As a second step technological instructions are reduced to the first level in the form of statement body.

As a next example we mention the implementation of the assembly language called SLAGH. The purpose of the implementation is twofold. Defining the statements of the SLAGH language as augment statements it becomes possible to simulate SLAGH programs in the CHANGE system. On the other hand appropriately changing the target language of the TRANSLATE statement one can generate microprograms from SLAGH programs.

The last example mentioned is an application of CHANGE related to the input-output analysis of the general linear system models. Such a problem occurs, for example, during the design of the general linear electrical network. The problem is reduced to the combinatorial problem of finding a spanning forest in a graph with  $k$  roots fixed. The solution is based on



the mapping of the given graph to an equivalent configuration of processors [Kov 81].

#### References

- [Bas 75] Basile, V.R.: A Structured Approach to Language Design, Computer languages, Vol. 1. 255-273.
- [Kov 81] Kovács, I. and Pávó I.: The Implementation of a k-tree Generation in a Parallel Computing System, to be published in Acta Cybernetica
- [Leg 73] Legendi, T.: The CHANGE Language/Multiprocessor, Computer and Automation Institute, HAS, 7/1973, Budapest (in Hungarian)
- [Leg 80] Legendi, T.: CHANGE User's Manual, Computer and Automation Institute, HAS, CDC 3300 User's Manuals, 15/1980, Budapest (in Hungarian)
- [Nap 76] Napper, R.B.E. and Fisher, R.N.: ALEC - A User Extensible Scientific Programming Language, Computer Journal, Vol. 19. 25-31.
- [Nap 80] Napper, R.B.E. and Fisher, R.N.: RCC - A User Extensible Systems Implementation Language, Computer Journal Vol. 23. 212-223.
- [Rai 78] Raiha, K., Saarinen, M., Soisalo-Soininen, E. and Tienari, M.: The Compiler Writing System HLP (Helsinki Language Processor), Department of Computer Science, University of Helsinki, Report A-1978-2.
- [Sim 81] Simon, E., Zachar, Z., Gyimóthy, T. and Turán Gy.: A Self-extending Compiler Generator Based on Attribute Grammars, to be published in the Proceedings of the Programming Systems'81 Conference, J. von Neumann Society, Budapest (in Hungarian)
- [Sol 71] Solntseff, N. and Yezerski, A.: A Survey of Extensible Programming Language, McMaster University Hamilton, Technical Report, No. 71/7.
- [Tho 77] Thompson, C.D. and Kung, H.T.: Sorting on a Mesh-connected Parallel Computer, CACM, Vol. 20. 263-271.

- [Tuc 75] Tucker, A.: Very High-level Language design: A Viewpoint, Computer Languages, Vol. 1. 3-16.
- [Weg 76] Wegner, P.: Programming Languages. The First 25 Years, IEEE Transactions on Computers, Vol. C-25. 1207-1225.
- [Wgb 71] Wegbreit, B.: The ECL Programming System, U.S. Air Force Technical report, F-19628-68-C-0101.
- [Win 79] Winograd, T.: Beyond Programming Languages, CACM, Vol. 22. 391-401.





## EFFECTIVE LOGIC OF PROGRAMMING LANGUAGES

*Marek A. Suchenek*

Institute of Computer Science Warsaw Technical University  
Warsaw, Poland

### Abstract

An effective logic (i.e. with decidable notion of proof from decidable set of premises, and with the completeness property) for programming languages including conditional-assignment-with-goto instructions and first order declarations of data structure is presented in this paper. Usefulness of the logic in proving the total correctness and the stop properties of programs is demonstrated here.

### Introduction

A programmer, who has lost long hours in hunting subtle bugs in his own program, could really appreciate an ease, that seeming evidence, deceives our intuition with. In fact, common sense alone does not matter a lot against a finesse of traps, that the programming practice lays. So it is no surprise for me that as long as programmable computers exist (may be even a bit longer) one feverishly searches efficient weapons against errors.

Besides all various preventive measures taken in order to secure a functional correctness for programs, methods of post factum reliance enlargement, known by the name of verification, deserve the special mention. Most natural of them, testing, is based on the principle: "the proof of the pudding is in the eating". Though in many cases such a procedure seems quite sufficient, what more it even may be sometimes the only possible way of verification, the effective adequacy of the method nevertheless arose some doubts. "Testing may reveal only the presence but never the absence of errors" is the thesis, which for all its over-categorical form reflects fairly well a scepticism, the practice of experimental inspection of the correctness of programs meets with.

The mentioned distrust has given rise to efforts towards making use of deductive methods where experience did not yield the expected results. Since — as being claimed — an experiment ascertains by no means functional features of a non-trivial program, maybe it would be possible to prove them to be valid beyond a doubt. With such purposes in mind, the trend towards mathematized theory of programs — which the work of McCarthy [12] is an example of — was nothing unusual. But in spite of the fact, that Turing's machine is numbered among basic notions of mathematical logic since a long time, the phrase "beyond a doubt" with reference to the proof of the correctness of the program, turned out to be difficult to formalize.

One has relatively easily succeeded in specifying — as Hoare [6] did — a set of axioms and rules of inference, that were commonly admitted to be evidently valid. However, the set (see Andréka, Németi, Sain [18]) leaves much to be desired in respect of its completeness: there are undoubtedly correct programs, the correctness of which cannot be proved applying



exclusively Hoare's rules and axioms. As conclusions concerning features of programs, drawn from a finite number of tests successfully executed, would be sometimes over-hasty, in so far the abridgement of the meaning of "beyond a doubt" to "being formalizable within Hoare's system" has appeared to be too far prudence.

On the other hand, attempts to reduce the proof of the correctness of a program to a proof in the sense of classical logic by appropriate axiomatization of the notion of "program" (e.g. in Burstall [2]) did not meet with wide approbation, because of doubts about its exact consistency with general comprehension of that term. It can be found in [16] (foot-note 1) that the above question is not baseless, at least as it concerns [2].

The requirement of completeness of the searched formal system has born a necessity of the rigorous definition of program's semantics, which escaped the attention of even such recognized authorities, as Dijkstra [3], Hoare and Wirth [7] or Manna [11]. This problem has been resolved on the base of the so-called transition relation (due to Mazurkiewicz and Pawlak), which is usually interpreted as abstract processor of the program, that is realized by (hence Pawlak's machine). Properties of programs, including the correctness, have been consequently described as common features of corresponding transition relations. With such determined semantics, the completeness of the programming logic  $L$  is defined as follows: if a program  $P$  has a property expressed by a formula  $q(P)$ , then there is a formal proof of  $q(P)$  within  $L$ , and vice versa.

Till now, one succeeded in constructing a few complete deduction systems concerning programs.

The Algorithmic Logic of Salwicki [15] and the Dynamic Logic of Harel [5], Meyer and Pratt are the best known ones. However, these systems have some unpleasant features that restrict slightly their scope of applicability: they involve infinitely long proofs (e.g. Algorithmic Logic) or undecidable sets of axioms (e.g. Dynamic Logic). For this reason they do not yield the effective definition of the notion of "beyond a doubt" with reference to the proof of the program's correctness: the first one through the necessity of inspection of infinitely long proofs, and the second one through the lack for a way of deciding whether or not a sentence is an axiom. The reason of the above imperfection lies in an a priori assumption that data structure, which a program refers to, includes the standard model for arithmetics – the object by virtue of Gödel's theorem [4] unknowable, if Church's thesis (Barwise [1], chap A1) is believed. Therefore is no surprise that modes of inference about so comprehended programs have successfully resisted automation.

Do we indeed require a perfect computer to contain an implementation of all naturals with usual arithmetical operations? It is difficult to imagine, how the requirement could be formally stated, to say nothing about its unfeasibility. What is more, the whole arithmetics need not to be necessary in the set of operations of a given program. It is verisimilar instead, that quite different data structures will turn out to be useful in concrete applications, what is always connected with the necessity of identification of such structures. Furthermore, it seems not to



be a reason to distinguish some of them, as for example arithmetics, by release from such an obligation.

They are declarations that play the role of identifiers of the so-called abstract data structure. As they need not to be (and usually are not) unambiguous, there exist various realizations of data structures of the very same program. In this case the verification problem reduces to the ascertainment that the verified program acts accordingly to the author's expectations in all possible realizations of its data structure (cf [18]). It is easily seen, that testing has still less to do with efficiency here.

The notion of portable programming is related to this point of view, and is taken as principle in e.g. FORTRAN standard definition [14]. It arose as a side effect of the standardization process of high level languages: it turned out that it is sensible to demand a program to be constructed in a way, providing its correct behaviour under each of various compilers that meet all the *e x p l i c i t* requirements of a standard language definition.

So comprehended program's properties preserved under a class of data structure, were investigated in Kfoury [8]. A more radical standpoint can be found in Manna [10], where uninterpreted flow schemata with no explicit assumptions as to the data structure are considered. However, as far as I know, the methods of reasoning about such partially interpreted programs have been nowhere yet effectively formalized but in [16].

The present paper concerns the Computational Logic of [16]. This is a deduction system with finitary rules of inference (infinite proofs are no longer needed), decidable set of axioms • (there exists an effective decision procedure of being an axiom) with completeness property, which means that the notion of "beyond a doubt" in respect to the proof of a property, expressible in the language of Computational Logic (here called Programming Language) has been eventually defined. On the other hand, it is shown in the sequel, that interesting properties of programs are possible to be expressed in that language, and then Computational Logic proves useful in the process of program verification.

The paper is organized as follows:

**Section 1** is a description of the Programming Language. It is expressive enough to compilers of arbitrary real programming language be written in.

**Section 2** is devoted to semantics — the notion of realization of program is introduced there. Alike in practice, where there may exist many essentially different realizations of a given program, or there may be no such realization at all, in the paper — unlike in the most of known theoretic solutions — the principle of the unique determination of semantics by the program itself is *n o t* assumed.



**Section 3** contains examples of program's properties. They concern the total correctness and halting problems.

**Section 4** contains results that show that these properties are expressible in Programming Language with satisfactory accuracy.

**Section 5** contains the definition of Computational Logic and the completeness theorem. The proof of the theorem, considering its size, is not presented here — it was published in [16].

**Section 6** contains the definition of the instruction of the Programming Language. The instructions are defined to be formulae of that language, similarly as in Kröger [9]. But unlike ours, the logic of [9] (Logic for Algorithmical Reasoning) is infinitary one.

**Section 7** consists of notes on restricted semantics. It is pointed out, that the rejection of some classes of realizations, e.g. the ones having too large computational power, has no influence on the completeness property of Computational Logic.

## 1. Programming Language

A *program* is a set of instructions and declarations. *Instructions* are conditional-assignment-with-goto instructions of the form

1.1.  $\alpha : \text{if } p \text{ then } x_1, \dots, x_n := t_1, \dots, t_n \text{ and go to } \beta$

where

- $\alpha$  and  $\beta$  are of naturals (*labels*):  $\alpha$  labels 1.1 and  $\beta$  points the instructions, that may be executed after the assignment of 1.1,
- $p$  is a formula (a logical expression) being the necessary condition for the assignment of 1.1.
- $x_1, \dots, x_n$  are variables:  $x_i$  is nonidentical with  $x_j$ , whenever  $i \neq j$ ,
- $t_1, \dots, t_n$  are terms (arithmetical expressions).

**Declarations** are first-order sentences, i.e. formulae without variables being not in the scope of a quantifier. E.g.:

$$\forall x \quad \exists y (x < y)$$

is the declaration, while  $\exists y (x < y)$  is not. It has the meaning: "for each  $x$  there is  $y$ , that  $x < y$ ".

Instructions describe actions, that have to be executed, whereas declarations define a



class of data structure, which the actions refer to. In the sequel, programs will be usually denoted by  $P, P'$ , etc, sets of instructions — by  $Q, Q', \dots$ , and sets of declarations — by  $R, R'$ , and so on. From the definition of program we shall assume that

$$P = Q \cup R, P' = Q' \cup R', \dots \text{ etc.}$$

## 2. Semantics of Programs

Let us assume the set of variables to be potentially infinite (virtual memory). A state of memory is uniquely determined by an infinite sequence. Elements of the sequence are the values of consecutive variables. The sequences will be denoted by  $\varphi, \psi$ , etc. We shall call them *valuations*.

A state of program's execution is uniquely determined by an instruction counter and a state of memory. Pairs  $\langle \alpha, \varphi \rangle$ , where  $\alpha$  is a label and  $\varphi$  is a valuation, encode this information, and will be called *situations*.

In order to execute the process described by a program, it suffices to construct an automaton, performing all the transitions between situations, according to that program. Such an automaton is called a *transition relation* (denoted by  $C, C'$ , etc.) and is a binary and transitive<sup>1)</sup> relation in the set of all situations. Being of two situations  $s_1$  and  $s_2$  in the transition relation is interpreted as existence of path (i.e. a sequence of consecutive transitions) from  $s_1$  to  $s_2$ . The transitivity of the relation means: if there are paths from  $s_1$  to  $s_2$  and from  $s_2$  to  $s_3$ , then there is a path from  $s_1$  to  $s_3$ .

A *computational system* is a triple

2.1.  $\langle A, C, F \rangle$

usually denoted as  $\mathcal{H}_c$ , where

- $A$  is the set of all values (of variables) available in the system  $\mathcal{H}_c$ ,
- $C$  is a transition relation; valuations embedded in  $C$  must be sequences of elements of  $A$ ,
- $F$  is a sequence of relations-and-functions which are standard ones in  $\mathcal{H}_c$ ; e.g. in  $F$  may appear: zeroing, incrementation, comparison with zero, etc.; the elements of  $F$  are the only ones, whose names (symbols) may appear in programs.

1)  $C$  is a binary transitive relation if  $\langle s_1, s_2 \rangle \in C$  and  $\langle s_2, s_3 \rangle \in C$  implies  $\langle s_1, s_3 \rangle \in C$ .



If  $q$  is an instruction of the form 1.1, then  $\mathcal{M}_c$  satisfies (may be among others)  $q$  iff for each valuation  $\varphi$  with values in  $A$  the condition:

if  $p$  is true when evaluated in  $\mathcal{M}_c$  by  $\varphi$  (denotation  $\varphi^*(p) = \text{true}$ ) then the transition  $(\langle \alpha, \varphi \rangle, \langle \beta, \psi \rangle)$  is in  $C$  denotation:  $(\langle \alpha, \varphi \rangle, \langle \beta, \psi \rangle) \in C$

holds, where  $\psi$  is defined by equation

$$2.2. \quad \psi(x) = \begin{cases} \varphi(x) & \text{if } x \notin \{x_1, \dots, x_n\} \\ \varphi^*(t_i) & \text{if } x \text{ is identical with } x_i \end{cases}$$

and  $\varphi^*(t_i)$  denotes the value of term  $t_i$  evaluated  $\mathcal{M}_c$  by  $\varphi$ . It is easily seen, that the notion of satisfaction of instruction is an explicitness of its intuitively comprehended execution. In the sequel, the denotation  $\mathcal{M}_c \models q$  will be applied instead of:  $\mathcal{M}_c$  satisfies  $q$ .

If  $r$  is a declaration, then  $\mathcal{M}_c$  satisfies  $r$  if  $r$  is true in the structure  $\langle A, F \rangle$  (cf. Barwise [1], chap. A1). In that case we shall write  $\mathcal{M}_c \models r$ .

If  $P$  is a program, then  $\mathcal{M}_c$  is a realization of  $P$  (denotation:  $\mathcal{M}_c \models P$ ) if for  $\forall p \in P$ ,  $\mathcal{M}_c \models p$  holds. It means that  $\mathcal{M}_c \models P$  if  $C$  realizes (at least) the executions of instructions of  $P$ , nad  $\langle A, F \rangle$  is a structure described by declarations of  $R$ . A program  $P$  and a structure  $\langle A, F \rangle \models R$  need not uniquely determine the realization  $\langle A, C, F \rangle$ , since, as it follows from the definition of realization, the condition:

if  $\langle A, C, F \rangle \models P$  and  $C \subseteq C'$  is a transition relation in  $A$ , then  
 $\langle A, C', F \rangle \models P$

holds.

If  $P$  is a program then a computational system  $\langle A, C, F \rangle$  is an *implementation* of  $P$  iff  $\langle A, C, F \rangle \models P$  and  $C \subseteq C'$  whenever  $\langle A, C', F \rangle \models P$ . In other words, an implementation of a program is its minimal (in the sense of inclusion of transition relations) realization. So the transition of an implementation encodes exactly the actions specified by a program.

A program  $P$  and a structure  $\langle A, F \rangle \models R$  uniquely determine  $P$ 's implementation, which is a computational system  $\langle A, C, F \rangle$  where  $C$  is the transitive closure <sup>2)</sup> of the so-called direct transition relation of  $P$  in  $\langle A, F \rangle$ . However, it does not mean, that a program itself uniquely determines its implementation.

2) the least transitive relation which includes a relation in question



The defined syntax and semantics enable to deal with concurrent programs. If each instruction is thought to be an atomic and instantaneous one, what is widely accepted – cf. Pnueli [13], then the processor of the program may be considered as a set of independent atomic processors. Each of them corresponds to one and only one instruction and acts as follows:

- (1) At the very beginning all the processors corresponding to instructions labelled by the entry label are active, while the rest of them are passive ones.
- (2) The processor corresponding to 1.1 performs the specified assignement only if  $p$  is true, and the processor is active, and no other processor is acting at the moment.
- (3) If the label  $\alpha$  is different from the label  $\beta$ , then the processor corresponding to 1.1. turns all the processors corresponding to instructions labelled by  $\beta$  active, and all corresponding to ones labelled by  $\alpha$  passive – simultanneously with the performed assignement.
- (4) The execution of the program is finished iff there are no active processors.

Such an approach to concurrency is known by the name of subjective semantics, and is based on expressing the concurrency in the terms of non-determinism (see e.g. Winkowski [19] for details).

**Example** The program (11 is always true formula)

```
{  $\neg(0 = 1)$ ;
 $\alpha$  : if 11 then  $x_1, x_2 := 0, 0$  and  $\beta$ ;
 $\beta$  : if  $x_1 = 0$  then  $x_1, y_1 := 1, f(y_1)$  and  $\beta$ ;
 $\beta$  : if  $x_2 = 0$  then  $x_2, y_2 := 1, f(y_2)$  and  $\beta$ ;
 $\beta$  : if  $(x_1 = 1) \ \& \ (x_2 = 1)$  then  $z := g(y_1, y_2)$  and  $\gamma$  }
```

"computes" a value of the term  $g(f(y_1), f(y_2))$  with values of  $f(y_1)$  and  $f(y_2)$  being computed concurrently. ■

If  $P$  and  $P'$  are programs, then  $P$  *semantically implies*  $P'$  iff each realization of  $P$  is a realization of  $P'$  too (i.e. for all  $\mathcal{U}_c$ ,  $\mathcal{U}_c \models P'$  whenever  $\mathcal{U}_c \models P$ ). This fact is denoted by  $P \models P'$  and means that  $P$  "performs" at least actions, which  $P'$  "performs".

**Example** If  $P' \subseteq P$ , then – by virtue of the definition of realization –  $P \models P'$ . ■



**Example.**

$\{ \alpha : \text{ if } \neg(x = 0) \text{ then } x := f(x) \text{ and } \alpha;$

$\alpha : \text{ if } x = 0 \text{ then } x := g(x) \text{ and } \beta \} \models$

$\alpha : \text{ if } \neg(x = 0) \ \& \ f(x) = 0 \text{ then } x := g(f(x)) \text{ and } \beta$

The opposite implication does not hold. ■

### 3. Properties of Programs

Let  $\alpha$  be the entry label and  $\beta$  the exit one. We shall deal with the following properties of realizations of the program  $P$ :

3.1. *Property* For the values of "input" variables  $x_1, \dots, x_n$ , which the condition  $p(x_1, \dots, x_n)$  holds for, each realization of  $P$  computes the values of "output" variables:  $y_1 = f_1(x_1, \dots, x_n), \dots, y_m = f_m(x_1, \dots, x_n)$ , and halts.

3.2. *Property* There is a path from  $\alpha$  to  $\beta$  in some implementation  $\mathcal{M}_c$  of  $P$ , for some initial valuation  $\varphi$  and some terminal valuation  $\psi$  in  $\mathcal{M}_c$ .

3.3. *Property* For all implementations  $\mathcal{M}_c$  of  $P$  and for all initial valuations  $\varphi$  in  $\mathcal{M}_c$ , there is a path from  $\alpha$  to  $\beta$ .

Let *route* in  $\langle A, C, F \rangle$  denote a sequence of situations  $\langle s_1, \dots, s_n, \dots \rangle$ , such that  $\langle s_i, s_{i+1} \rangle \in C$  for  $i = 1, \dots, n-1, \dots$ , and  $s$  is the last situation of a route only if for no  $s', \langle s, s' \rangle \in C$  holds.

3.4. *Property* Each route in arbitrary implementation  $\mathcal{M}_c$  of program  $P$  ends at  $\beta$ , whenever starts at  $\alpha$ .

The property 3.1. is a weak version of total correctness. The properties 3.2 – 3.4. are different versions of stop property, with 3.4 being the strongest one.

### 4. The Expressiveness of the Properties.

The property 3.1. is immediately expressible in Programming Language.

4.1. *Theorem* A program  $P$  is of the property 3.1. iff

$P \models \alpha : \text{ if } p(x_1, \dots, x_n) \text{ then } y_1, \dots, y_m := f_1(x_1, \dots, x_n), \dots,$   
 $f_m(x_1, \dots, x_n) \text{ and } \beta.$



*Proof* (instant) from the definition of realization. ■

In order to express rigorously the properties 3.2 – 3.4, one needs a language, which is a proper extension of our Programming Language <sup>3)</sup>. However, one can "interpolate" the properties with satisfactory accuracy in Programming Language.

4.2. *Theorem* If  $P$  is of the property 3.2, then there is an instruction  $q$  of the form 1.1, that

$$\mathcal{Q} \models q$$

$$\varphi^*(p) = \text{true} \quad (p \text{ is satisfied by } \varphi \text{ in } \mathcal{M}_t)$$

$\psi$  is defined by 2.2.

*Proof* in [17]. ■

By other words, if there is a path from  $\alpha$  to  $\beta$  in some implementation of  $P$ , then  $\mathcal{Q}$  semantically implies an instruction, that forces that path existence.

4.3. *Theorem* If  $P$  is of the property 3.3, then there is a set  $P'$  of instructions

$$P' = \{ \alpha : \text{if } p_1 \text{ then } \dots \text{ and } \beta; \dots ; \alpha : \text{if } p_j \text{ then } \dots \text{ and } \beta \}$$

that

$$P \models P'$$

$$R \models p_1 \vee \dots \vee p_j \quad (\text{the } p_i\text{'s exhaust all possible cases})$$

$$R \models \neg (p_i \& p_k) \quad \text{for each } i \neq k \quad (\text{the } p_i\text{'s are disjoint}).$$

*Proof* in [17]. ■

If one has to deal with programs including infinite sets of instructions then the property 3.4. is beyond an effective investigation. Indeed, each of instructions of the program may be responsible for lack of the property, so in order to ascertain that it holds, one needs to consider all of the infinite sets of instructions.

4.4 *Theorem* If  $P$  is of the property 3.4. and  $\mathcal{Q}$  is finite, then there is a set  $P'$  of the form as in the 4.3. theorem, that

$$P \models P'$$

$$R \models p_1 \vee \dots \vee p_j$$

if  $(\langle \alpha, \varphi \rangle, \langle \beta, \psi \rangle) \in C$  then for some instruction  $q \in P'$  of the form 1.1,  $\varphi^*(p) = \text{true}$  and  $\psi$  is given by 2.2 (i.e. if there is a path from  $\alpha$  to  $\beta$  in an implementation of  $P$

<sup>3)</sup> The needed language must not have a partially decidable set of tautologies.



then some element of  $P'$  forces the existence of that

*Proof* in [17]. ■

From the above theorem it follows, that if a finite program  $P$  is of the property 3.4 then there is a finite loopless program  $P'$ , that is equivalent to  $P$  in respect to input-output characteristics.

## 5. Computational Logic

One may pose the question, whether the semantical implication is syntactically characterizable or not; i.e. whether there is a set of axioms and rules of inference, that  $P \models q$  is equivalent to existence of proof of  $q$  from  $P$  treated as set of premises (denotation:  $P \models q$ ).

The answer is "yes", if one does not impose any extra requirements. It suffices to take the set

$$\{ p \mid p \text{ is an instruction or declaration and } \emptyset \models p \}$$

as set of axioms, and the set

$$\left\{ \frac{P}{p} \mid P \text{ is a program, } p \text{ is an instruction or declaration and } P \models p \right\}$$

as set of rules of inference. However, the logic defined this way, need not be effective. For the effectiveness of the logic, the set of proofs should be decidable, unless the set of premises is undecidable. In other words, having a sequence of formulae and a decidable set of premises we should be able to decide by inspection, whether it is a proof or not. The Computational Logic satisfies this extra requirement.

Let  $L$  be a first-order language and let  $T$  denote the set of all terms of  $L$ . By *substitution* (in  $L$ ) we mean each function

$$\sigma : T \rightarrow T$$

satisfying:

- if  $f$  is  $n$ -ary function symbol and  $t_1, \dots, t_n$  are in  $T$ , then  $\sigma[ft_1 \dots t_n]$  is identical with  $f\sigma[t_1] \dots \sigma[t_n]$ ,
- if  $c$  is a constant symbol then  $\sigma[c]$  is identical with  $c$ , and
- there are only finitely many variables  $x$  in  $L$  that  $\sigma[x]$  is non-identical with  $x$ , where  $\sigma[t]$  denotes the value of  $\sigma$  on  $t$ .

The *languages*  $M$  we shall concentrate on, are those of first-order — with the equality symbol- which meet the conditions:

(1) the set of all terms of  $M$  is decidable, and

(2) the set of all realation symbols of  $M$  includes a decidable subset  $S$ , that for each number  $n$  there are infinitely many  $n$ -ary symbols in  $S$ .

By the conditions (1) and (2) there is such a  $1 - 1$  recursive mapping from the set of all sequences of the form

(3)  $\langle \sigma, y_1, \dots, y_n, \alpha, \beta \rangle$

(where  $\sigma$  is a substitution,  $y_1, \dots, y_n$  are variables,  $n, \alpha, \beta$  are naturals) onto  $S$ , that the arity of a symbol corresponding to (3) is  $n$ . For example of such a mapping see [16] (foot-note 2).

Further on, we shall fix the language and the mapping, denoting the *symbol* corresponding to (3) by

$sub(\sigma, y_1, \dots, y_n, \alpha, \beta)$

Now we state the set of *axioms* of Computational Logic. First, these are all formulae of forms listed below. (The parenthesisless notation is applied in relation to function-and-relation symbols).

(A1)  $p \supset (q \supset p)$

(A2)  $[p \supset (q \supset r)] \supset [(p \supset p) \supset (p \supset r)]$

(A3)  $[p \supset 0] \supset 0 \supset p$

(A4)  $\forall x(p \supset q) \supset (p \supset \forall x q)$  provided  $x$  is not free in  $p$

(A5)  $\forall x p \supset p'$

where  $p, q, r$  denote a formulae of  $M$ ,  $0$  is the falsehood symbol,  $x$  is a variable and  $p'$  is a formula obtained from  $p$  by putting a term  $t$  instead of each free occurrence of  $x$ , provided no occurrence of  $t$  so introduced contains the bound occurrences of a variable in  $p'$ .

(I1)  $t = t$  for every term  $t$  of  $M$

(I2)  $t_i = s_i \supset f t_1 \dots t_n = f t_1 \dots t_{i-1} s_i t_{i+1} \dots t_n$  for every  $n$ -ary function symbol of  $M$  and  $i = 1, \dots, n$



- (I3)  $t_i = s_i \supset (rt_1 \dots t_n \supset rt_1 \dots t_{i-1} s_i t_{i+1} \dots t_n)$  for every  $n$ -ary relation symbol of  $M$  and  $i = 1, \dots, n$ .

Second, these are all universal closures of formulae of forms listed below.

- (T1)  $sub(\sigma, y_1, \dots, y_n, \alpha, \beta) x_1 \dots x_n \supset$   
 $\supset sub(\sigma, y_{\pi(1)}, \dots, y_{\pi(n)}, \alpha, \beta) x_{\pi(1)} \dots x_{\pi(n)}$   
 where  $\pi$  denotes a permutation on  $\{1, \dots, n\}$
- (T2)  $\forall y sub(\sigma, y_1, \dots, y_n, y_{n+1}, \alpha, \beta) x_1 \dots x_n y \equiv$   
 $\equiv sub(\sigma, y_1, \dots, y_n, \alpha, \beta) x_1 \dots x_n$   
 for  $y_{n+1} \notin \{y_1, \dots, y_n\}$  and  $y \notin \{x_1, \dots, x_n\}$
- (T3)  $sub(\sigma, y_1, \dots, y_n, y_n, \alpha, \beta) x_1 \dots x_n y \equiv$   
 $\equiv (x_n = y \supset sub(\sigma, y_1, \dots, y_n, \alpha, \beta) x_1 \dots x_n)$
- (T4)  $(\sigma[z_1] = \tau[z_1] \& \dots \& \sigma[z_k] = \tau[z_k] \&$   
 $\& sub(\sigma, y_1, \dots, y_n, \alpha, \beta) x_1 \dots x_n) \supset$   
 $\supset sub(\tau, y_1, \dots, y_n, \alpha, \beta) x_1 \dots x_n$

where  $\sigma$  and  $\tau$  are substitutions identical on variables other than  $z_1, \dots, z_k$ , and  $x_1, \dots, x_n$  are the all variables of the terms  $\sigma[z_i], \tau[z_i]$  ( $i = 1, \dots, k$ )

- (T5)  $(sub(\sigma, y_1, \dots, y_n, \alpha, \beta) x_1 \dots x_n \&$   
 $\& sub(\tau, y_1, \dots, y_n, \beta, \gamma) \sigma[x_1] \dots \sigma[x_n]) \supset$   
 $\supset sub(\sigma\tau, y_1, \dots, y_n, \alpha, \gamma) x_1 \dots x_n$

provided the set of all variables of the terms  $\sigma[x_n]$  is included in  $\{x_1, \dots, x_n\}$ ,  $\sigma\tau$  is the composition of  $\sigma$  and  $\tau : \sigma\tau[t] = \sigma[\tau[t]]$ .

The only *rules of inference* are

$$\frac{p, p \supset q}{q} \text{ (modus ponens)}$$

$$\frac{p}{\forall y p'} \text{ (generalization)}$$

where  $p$  and  $q$  denote formulae of  $M$ , and  $p'$  denotes the formula obtained from  $p$  by putting the variable  $y$  instead of each free occurrence of  $x$ , provided no occurrence of  $y$  so introduced is bound in  $p'$ .

The notion of *proof* is defined as usual (Barwise [1], chap. A1). We write  $\Sigma \vdash p$  if only if there is a proof of  $p$  from a subset of  $\Sigma$ .

It is quite obvious that the set of axioms of the Computational Logic is decidable unless the set of these relation symbols of  $M$  which are not  $S$  is undecidable.

In order to define the semantics of the language  $M$ , we take advantage of the computational structure 2.1.

We let the terms and formulae of  $M \setminus S$  maintain in  $\mathcal{H}_c$  their ordinary first-order semantics (for details see Barwise [1], chap. A1), writing

$$\langle \mathcal{H}_c, \varphi \rangle \models p$$

for  $\langle A, F \rangle \models p[\varphi]$ , where  $\varphi$  denotes an interpretation of  $M \setminus S$  in  $\langle A, F \rangle$ . For the atomic formulae of  $S$  we define

$$\langle \mathcal{H}_c, \varphi \rangle \models \text{sub}(\sigma, y_1, \dots, y_n, \alpha, \beta) t_1 \dots t_n$$

to be valid if and only if the following implication holds: for every interpretation  $\psi$  in  $\mathcal{C}_c$  satisfying  $\psi(y_1) = \varphi(t_1)$  and  $\dots$  and  $\psi(y_n) = \varphi(t_n)$  the pair  $(\langle \alpha, \psi \rangle, \langle \beta, \psi, \sigma \rangle)$  is in  $C$  where  $\psi\sigma$  denotes such an interpretation in  $\mathcal{H}$  that for each term  $t$  of  $M$  there is  $\psi\sigma(t) = \psi(\sigma[t])$ .

Finally, by the usual induction we extend the (4) over the set of all formulae of  $M$ .

We shall write  $\mathcal{H}_c \models p$  whenever  $\langle \mathcal{H}_c, \varphi \rangle \models p$  holds for every  $\varphi$  in  $\mathcal{H}_c$ .

Note that given  $\varphi$ , the formula  $\text{sub}(\sigma, y_1, \dots, y_n, \alpha, \beta) t_1 \dots t_n$  has exactly the meaning of the instruction: if the value of  $y_i$  equals to  $\varphi(t_i)$  for  $i = 1, \dots, n$ , perform the substitution  $\sigma$  and go to  $\beta$ .

The following theorem is essential for our Computational Logic:



**5.1. The Completeness Theorem** For every formula  $p$  and every set  $\Sigma$  of sentences of  $M$ , the conditions,

$$\Sigma \models p \text{ and } \Sigma \vdash p$$

are equivalent.

**Proof** in [16]. ■

The Computational Logic satisfies the Löwenheim-Skolem-Tarski and the Compactness Theorem, hence by virtue of the Lindström Theorem (see Barwise [1], chap. A1) it is of the first-order character.

## 6. Instructions

Let  $p(y_1, \dots, y_n)$  denote such a formula of  $M$  that all free variables of  $p$  are in  $\{y_1, \dots, y_n\}$ , and no symbol of  $S$  occurs in  $p$ . Let  $\sigma$  be a substitution being the identity outside of  $\{x_1, \dots, x_k\}$ . We apply

$$\alpha : \text{if } p(y_1, \dots, y_n) \text{ then } x_1, \dots, x_k := \sigma[x_1], \dots, \sigma[x_k] \text{ and } \beta$$

or simply

$$\alpha : \text{if } p \text{ then } \sigma \text{ and } \beta$$

as abbreviation of

$$\forall y_1 \dots \forall y_n (p(y_1, \dots, y_n) \supset \text{sub}(\sigma, y_1, \dots, y_n, \alpha, \beta) y_1 \dots y_n)$$

and call it an *instruction*. It is easily seen that

$$\langle A, C, F \rangle \models \alpha : \text{if } p \text{ then } \sigma \text{ and } \beta$$

if and only if for arbitrary interpretation  $\varphi$  in  $A$ ,  $\varphi(p) = \text{true}$  implies  $(\langle \alpha, \varphi \rangle, \langle \beta, \varphi \sigma \rangle) \in C$ , thus the defined above instructions have their ordinary meaning. Hence the defined language  $M$  is our Programming Language of Section 1.

## 7. Restricted semantics

In the class of computational systems previously defined, there are ones, that perform actions being inaccessible for any program. Let  $K$  denote the class of all implementations having finite or countable universe:

$K = \{ \langle A, C, F \rangle \mid \text{card}(A) \leq \rho_0 \text{ and } \langle A, C, F \rangle \text{ is an implementation of some program} \}$

We define the relation  $\models_0$  of *semantical implication in restricted sense* as follows:

$P \models_0 q$  iff for each  $\mathcal{M}_c \in K$ ,  $\mathcal{M}_c \models P$  implies  $\mathcal{M}_c \models q$ . However, these measures have no influence on the completeness theorem of our logic.

**7.1. Theorem.** For the arbitrary countable program  $P$  and formula  $q$ , the conditions  $P \models q$  and  $P \models_0 q$  are equivalent.

**Proof** in the English version of [16]. ■

**Acknowledgement.** I am greatly indebted to professor Andrzej Salwicki for reading an earlier version of the manuscript and making many helpful comments.

## Reference

- [1] Barwise, J. (editor): "Handbook of Mathematical Logic", North-Holland, Amsterdam 1978.
- [2] Burstall, R.M.: "Formal Description of Program Structure and Semantics in First Order Logic", Machine Intelligence 5(1969), pp. 79-98.
- [3] Dijkstra, E.W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Comm A. C. M. 18(1975), pp 453-457.
- [4] Gödel, K.: "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme", Monatsh. Math. Phys. 38 (1931), pp. 173-198.
- [5] Harel, D.: "Arithmetical Completeness in Logics of Programs" in "Automata, Languages and Programming. Proc. of 5-th Coll.", Lecture Notes in Computer Science 62 (1978), pp 268-288.
- [6] Hoare, C.A.R.: "An Axiomatic Basis for Computer Programming" Comm. ACM 12 (1969), pp. 576-580.
- [7] Hoare, C.A.R., Wirth, N.: "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica 2(1973), pp. 335-355.
- [8] Kfoury, A.J.: "Comparing Algebraic Structures up to Algorithmic Equivalence" in Nivat, M. (editor): "Automata, Languages and Programming, Proc. IRIA", North-Holland, Amsterdam 1973.



- [9] Kröger, F.: "LAR: A Logic of Algorithmic Reasoning", Acta Informatica 8 (1977), pp. 243-266.
- [10] Manna, Z.: "Mathematical Theory of Computation", Mc Graw-Hill, New-York 1974.
- [11] Manna, Z., Pnueli, A.: "Axiomatic Approach to Total Correctness of Programs", Acta Informatica 3 (1974), pp 243-263.
- [12] McCarthy, J.: "A Basis for a Mathematical Theory of Computation", in "Computer Programming and Formal Systems", North-Holland, Amsterdam 1963.
- [13] Pnueli, A.: "The Temporal Semantics of Concurrent Programs" in "Semantics of Concurrent Computation", Lecture Notes in Computer Science 70 (1979) pp 1-20.
- [14] Programming Language FORTRAN, ISO Recommendation R1539, 1972.
- [15] Salwicki, A.: "Formalized Algorithmic Languages", Bull. Acad. Polon. Sci. Ser. Sci. Math. Astronom. Phys. 18 (1970), pp/ 227-232.
- [16] Suchenek, M.: "Notes on Computational Logic", Thesis, Warsaw Technical University, 1978, in Polish. (Slightly shortened English version appeared as Research Report No 11, Inst. of Comp. Sc., Warsaw Technical University 1980).
- [17] Suchenek, M.: "Is First Order Logic Really Useless in Proofs of Interesting Properties of Programs?" (to appear).
- [18] H. Andréka, I. Németi, I. Sain: "Completeness problems in verification of programs and program schemes" in MFCS'79, Lecture Notes in Computer Science vol. 74., Springer-Verlag 1979. pp. 208-219
- [19] Winkowski, J.: "A formalism for describing non-sequential processes" Fundamenta Informaticae 2 (1978) pp. 129-139.



C 27196

